```
// Computer Program Listing Appendix Under 37 CFR 1.52(e)
// Match.txt
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
package com.sybase.patriotact.utils;
/*
*/
/* import list */
/*
import org.apache.log4j.Category;
import org.apache.log4j.PropertyConfigurator;
import com.sybase.utils.generic.GenericUtilityTools;
import java.util.PropertyResourceBundle;
import java.util.ResourceBundle;
//import com.sybase.utils.generic.LOG4JLogger;
*/
import java.util.Vector;
import java.io.PrintWriter;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.PrintWriter;
//Log4j
//import org.apache.log4j.Logger;
public class Match
{
   //private static Logger _log = Logger.getLogger(Match.class); //Log4j initializer
   /*
    * Does a calculates a score between two strings.
    * @param inString1
    * @param inString2
    * @since 1.0
    */
//   private static final String VOWELS = "AEIOU" ;
   private static final String FRONTV = "EIY"   ;
   private static final String VARSON = "CSPTG" ;
   private static final int MAX_CODE_LENGTH = 4 ;
   private static final String VOWELS = "AEIOUY";
   //skip these when at start of word: "GN", "KN", "PN", "WR", "PS"
   private static final String[] WORD_START_SKIP_1 = {"GN", "KN", "PN", "WR", "PS"};
   //various germanic
   private static final String[] C_GERMANIC = {"BACHER", "MACHER"};
   //greek roots e.g. 'chemistry', 'chorus'
   private static final String[] C_GREEK = {"HARAC", "HARIS", "HOR", "HYM", "HIA", "HEM"};
   //germanic, greek, or otherwise 'ch' for 'kh' sound
   private static final String[] GERMANIC = {"VAN ", "VON ", "SCH"};
   // 'architect but not 'arch', 'orchestra', 'orchid'
   private static final String[] C_RCH= {"ORCHES", "ARCHIT", "ORCHID"} ;
   private static final String[] C_TS= {"T", "S"} ;
   private static final String[] C_AOUE= {"A", "O", "U", "E"} ;
```

```
//e.g., 'wachtler', 'wechsler', but not 'tichner'
private static final String[] C_LRNMBHFVW= {"L", "R", "N", "M", "B", "H", "F", "V", "W", " "} ;
private static final String[] C_IEH= {"I", "E", "H"} ;
//'accident', 'accede' 'succeed'
private static final String[] C_UCC= {"UCCEE", "UCCES"};
private static final String[] C_CKCGCQ= {"CK", "CG", "CQ"} ;
private static final String[] C_CICECY= {"CI", "CE", "CY"} ;
//italian vs. english
private static final String[] C_CIOCIECIA= {"CIO", "CIE", "CIA"} ;
private static final String[] C_CQG= {"C", "Q", "G"} ;
private static final String[] C_CKQ= {"C", "K", "Q"} ;
private static final String[] C_CECI= {"CE", "CI"} ;
private static final String[] D_IEY= {"I", "E", "Y"} ;
private static final String[] D_DTDD= {"DT", "DD"} ;
private static final String[] G_BHD= {"B", "H", "D"} ;
private static final String[] G_BH= {"B", "H"} ;
//e.g., 'laugh', 'McLaughlin', 'cough', 'gough', 'rough', 'tough'
private static final String[] G_CGLRT= {"C", "G", "L", "R", "T"} ;
//-ges-,-gep-,-gel-, -gie- at beginning
private static final String[] G_GES_GEP= {"ES", "EP", "EB", "EL", "EY", "IB", "IL", "IN", "IE", "EI", "ER"} ;
// -ger-,  -gy-
private static final String[] G_GER_GY= {"DANGER", "RANGER", "MANGER"} ;
private static final String[] G_EI= {"E", "I"} ;
private static final String[] G_RGYOGY= {"RGY", "OGY"} ;
// italian e.g, 'biaggi'
private static final String[] G_EIY= {"E", "I", "Y"} ;
private static final String[] G_ITALIAN= {"AGGI", "OGGI"} ;
private static final String[] J_LTKSNMBZ= {"L", "T", "K", "S", "N", "M", "B", "Z"} ;
private static final String[] J_SKL= {"S", "K", "L"} ;
private static final String[] L_SPANISH= {"ILLO", "ILLA", "ALLE"} ;
private static final String[] L_ASOS= {"AS", "OS"} ;
private static final String[] L_AO= {"A", "O"} ;
private static final String[] P_PB= {"P", "B"} ;
private static final String[] R_MEMA= {"ME", "MA"} ;
//special cases 'island', 'isle', 'carlisle', 'carlysle'
private static final String[] S_ISLYSL= {"ISL", "YSL"} ;
private static final String[] S_GERMANIC= {"HEIM", "HOEK", "HOLM", "HOLZ"} ;
//italian & armenian
private static final String[] S_ITALIAN= {"SIO", "SIA", "SIAN"} ;
private static final String[] S_MNLW= {"M", "N", "L", "W"} ;
//dutch origin, e.g. 'school', 'schooner'
private static final String[] S_DUTCH= {"OO", "ER", "EN", "UY", "ED", "EM"} ;
//'schermerhorn', 'schenker'
private static final String[] S_DUTCH_2= {"ER", "EN"} ;
//french e.g. 'resnais', 'artois'
private static final String[] S_FRENCH= {"AI", "OI"} ;
private static final String[] S_SZ= {"S", "Z"} ;
private static final String[] T_TIATCH= {"TIA", "TCH"} ;
//special case 'thomas', 'thames' or germanic
private static final String[] T_OMAM= { "OM", "AM"} ;
```

```
    private static final String[] T_TD= { "T", "D"} ;
    private static final String[] T_SLAVIC= { "EWSKI", "EWSKY", "OWSKI", "OWSKY"} ;
    //polish e.g. 'filipowicz'
    private static final String[] T_POLISH= { "WICZ", "WITZ"} ;
    private static final String[] X_IAUEAU= { "IAU", "EAU"} ;
    private static final String[] X_AUOU= { "AU", "OU"} ;
    private static final String[] X_CX= { "C", "X"} ;
    private static final String[] Z_ZOZIZA= { "ZO", "ZI", "ZA"} ;
//    private final static Category _cat = Category.getInstance(Match.class.getName());
//    protected GenericUtilityTools _wp = new GenericUtilityTools();
    private static String _configFile = Match.class.getName()+"Init";
    public Match()
    {
//        PropertyConfigurator.configure(_wp.getProperties(PropertyResourceBundle.getBundle(_configFile)));
//        _log.debug(_log.getName()+" Configured ...");
    }
    public static double score(String inString1, String inString2)
    {
        //_log.info("parameter to method : score : <" + inString1 + "> inString2 <" + inString2 + ">");
        //Get rid of any whitespace that may be on the strings
        // && drop the strings to lower case (nvr)
        //optimistic choice for match use the shortest string to search in the longest
        inString1 = inString1.trim().toLowerCase();
        inString2 = inString2.trim().toLowerCase();
        int inStringLen1 = inString1.length();
        int inStringLen2 = inString2.length();
  float leadingCharMatched = (float)0.55;
  float leadingCharTransposed = (float)1.0;
  //if (inStringLen1 < inStringLen2)
        if (inStringLen1 < inStringLen2)
        {
            while (inString1.length() < inStringLen2)
                {inString1 += " ";}
        }
        else if (inStringLen1 > inStringLen2)
        {
            while (inString2.length() < inStringLen1)
                {inString2 += " ";}
        }
        //The matching window is always half the length of the second string    rounded down
        int matchLength = inString2.length() / 2;
        int numberOfMatchingCharacters = 0;
        int numberOfTranspositions = 0;
        double score = 0;
        /*
        ** Optimization - if the two strings match exactly then don't    bother going any further, just return a
        ** value of 1.000.
        */
        if (inString1.equals(inString2))
        {
             score = 1.000;
        }
        else   //Do the Jaro-Winkler match
        {
            StringBuffer buf1 = new StringBuffer(inString1);
            for (int i = 0; i < buf1.length(); i++)
            {
                char c = buf1.charAt(i);
                //The start & end points roll with the position of the character in the first string
                int matchStart = i - (matchLength / 2);
                if (matchStart < 0)
                {
                     matchStart = 0;
                }
                int matchEnd = i + (matchLength / 2);
                if (matchEnd > inStringLen2)
                {
                    matchEnd = inStringLen2;
                }
                //_log.debug("Matching window from character " + matchStart + " to " + matchEnd);
                //First just check if it's at exactly the same position    in the second string, this avoids
                //the problem of false transpositions where it finds the correct character but at an earlier point
                int matchPoint = inString2.indexOf(c, i);
                if (matchPoint == i)
                {
                     numberOfMatchingCharacters++;
       if (matchPoint == 0) {
   leadingCharMatched = (float) leadingCharMatched + (float) 0.2;
       }
       else if (matchPoint == 1) {
   leadingCharMatched = (float) leadingCharMatched + (float) 0.15;
       }
       else if (matchPoint == 2) {
   leadingCharMatched = (float) leadingCharMatched + (float) 0.1;
       }
                     //_log.debug("Matching character : " + c + " found exactly at position " + matchPoint + " number of name
matches now " + numberOfMatchingCharacters);
                }
                else  //see if it's within the match window of    characters
                {
                    matchPoint = inString2.indexOf(c, matchStart);
                    //_log.debug("Matching character : " + c + " at position " + matchPoint);
                    //If the character is found
                    if (matchPoint >= 0)
                    {
                        if (matchPoint >= matchStart && matchPoint <= matchEnd)
                        {
                            numberOfMatchingCharacters++;
         if (matchPoint == 0) {
     leadingCharMatched = (float) leadingCharMatched + (float) 0.2;
```

```
        }
        else if (matchPoint == 1) {
    leadingCharMatched = (float) leadingCharMatched + (float) 0.15;
        }
        else if (matchPoint == 2) {
    leadingCharMatched = (float) leadingCharMatched + (float) 0.1;
        }
                        //_log.debug("Matching character : " + c +     " found at position " + matchPoint + " number of name
matches now " + numberOfMatchingCharacters);
                        //But if it's not in exactly the same place then it's a transposition
                        if (matchPoint != i)
                        {
                                numberOfTranspositions++;
            if (matchPoint == 0) {
    leadingCharTransposed = (float) leadingCharTransposed - (float) 0.2;
        }
            else if (matchPoint == 1) {
        leadingCharTransposed = (float) leadingCharTransposed - (float) 0.15;
            }
            else if (matchPoint == 2) {
        leadingCharTransposed = (float) leadingCharTransposed - (float) 0.1;
            }
                                //_log.debug("Transposition of characters in match, total transpositions now : " +
numberOfTranspositions);
                        }
                    }
                }
                else
                {
                        //If the letter was not found at all then    that's also counted as a transposition NOT nvr
                        //numberOfTranspositions++;
                }
            }
          }
          float nomChars = (float)numberOfMatchingCharacters;
          float score1 = 0 ;
          float score2 = 0 ;
          float score3 = 0  ;
    /*
    ** It is possible for leadingCharMatched to be more than one, e.g 'MEEK' v's 'MEK'. It's
    ** not a problem for this since it indicates duplicate letters appeared but make it 1.0
    ** to keep the scores total from being > 100.00.
    */
    if (leadingCharMatched > (float) 1.0) {
  leadingCharMatched = (float) 1.0;
    }
    if (nomChars==1)
          {
             score1 = (nomChars / (float)inStringLen1) * (float)0.5 * (float) leadingCharMatched;
             score2 = (nomChars / (float)inStringLen2) * (float)0.5 * (float) leadingCharMatched;
```

```
                score3 = (float)0.0;
                //_log.debug("nomChars: <"+nomChars+"> numberOfTranspositions: <"+numberOfTranspositions+">");
  //System.out.println("nomChars: <"+nomChars+"> numberOfTranspositions: <"+numberOfTranspositions+">");
            }
            else if (nomChars==2)
            {
                score1 = (nomChars / (float)inStringLen1) * (float)0.45 * (float) leadingCharMatched;
                score2 = (nomChars / (float)inStringLen2) * (float)0.45 * (float) leadingCharMatched;
                score3 = ((1 - ((float)numberOfTranspositions) / nomChars)) * (float)0.1 * (float) leadingCharTransposed;
                //_log.debug("nomChars: <"+nomChars+"> numberOfTranspositions: <"+numberOfTranspositions+">");
  //System.out.println("nomChars: <"+nomChars+"> numberOfTranspositions: <"+numberOfTranspositions+">");
            }
            else if (nomChars==3)
            {
                score1 = (nomChars / (float)inStringLen1) * (float)0.40 * (float) leadingCharMatched;
                score2 = (nomChars / (float)inStringLen2) * (float)0.40 * (float) leadingCharMatched;
                score3 = ((1 - ((float)numberOfTranspositions) / nomChars)) * (float)0.2 * (float) leadingCharTransposed;
                //_log.debug("nomChars: <"+nomChars+"> numberOfTranspositions: <"+numberOfTranspositions+">");
  //System.out.println("nomChars: <"+nomChars+"> numberOfTranspositions: <"+numberOfTranspositions+">");
            }
            else if (nomChars>3)
            {
                score1 = ((nomChars / (float)inStringLen1) * (float)0.333333) * (float) leadingCharMatched;
                score2 = ((nomChars / (float)inStringLen2) * (float)0.333333) * (float) leadingCharMatched;
                score3 = (((1 - ((float)numberOfTranspositions) / nomChars))) * (float)0.333334 * (float)
leadingCharTransposed;
  //System.out.println("********more than 3 chars matched " + score1 + " " + score2 + " " + score3 + " " +
leadingCharMatched + " " + leadingCharTransposed);
                //_log.debug("nomChars: <"+nomChars+"> numberOfTranspositions: <"+numberOfTranspositions+">");
  //System.out.println("nomChars: <"+nomChars+"> numberOfTranspositions: <"+numberOfTranspositions+">");
            }
            else
            {
                return 0;
            }
            score = score1 + score2 + score3;
            //_log.debug("score1: <" + score1+ "> score2: <"+score2+"> score3: <"+score3+">");
        }
 //adjust the score if the lengths of the two strings is greatly disparate
 if ((inStringLen1 < (inStringLen2-2)) || (inStringLen1 > (inStringLen2+2))) {
  score = score * 0.85;
 }
 //adjust the score if the lengths of the two strings is two characters
 else if ((inStringLen1 == (inStringLen2-2)) || (inStringLen1 == (inStringLen2+2))) {
  score = score * 0.925;
 }
        //_log.info("return from method : score : " + score +  "< matching >" + inString1 + "< to >" + inString2 + "<");
        //return (new Float((float)((score+.005f)*100f))).intValue()/100f;
        return (double)((score)*1000.0f);
    }
```

```
/*
public static String code( String txt)
{
   return code (txt, MAX_CODE_LENGTH);
}
public static String code( String txt , int codeSize)
{
   int mtsz = 0  ;
   boolean hard = false ;
   if(( txt == null ) ||
    ( txt.length() == 0 )) return "" ;
   // single character is itself
   if( txt.length() == 1 ) return txt.toUpperCase() ;
   //
   char[] inwd = txt.toUpperCase().toCharArray() ;
   //
   String tmpS ;
   StringBuffer local = new StringBuffer( 40 ); // manipulate
   StringBuffer code = new StringBuffer( 10 ) ; //   output
   // handle initial 2 characters exceptions
   switch( inwd[0] )
   {
      case 'K': case 'G' : case 'P' : // looking for KN, etc
         if( inwd[1] == 'N') local.append(inwd, 1, inwd.length - 1 );
         else local.append( inwd );
         break;
      case 'A': // looking for AE
         if( inwd[1] == 'E' ) local.append(inwd, 1, inwd.length - 1 );
         else local.append( inwd );
         break;
      case 'W' : // looking for WR or WH
         if( inwd[1] == 'R' )   // WR -> R
         {
            local.append(inwd, 1, inwd.length - 1 ); break ;
         }
         if( inwd[1] == 'H')
         {
            local.append(inwd, 1, inwd.length - 1 );
            local.setCharAt( 0,'W'); // WH -> W
         }
         else local.append( inwd );
         break;
      case 'X' : // initial X becomes S
         inwd[0] = 'S' ;local.append( inwd );
         break ;
      default :
         local.append( inwd );
   } // now local has working string with initials fixed
   int wdsz = local.length();
   int n = 0 ;
```

```
while((mtsz < codeSize ) && // max code size of 4 works well
    (n < wdsz ) )
{
    char symb = local.charAt(n) ;
    // remove duplicate letters except C
    if(( symb != 'C' ) &&
         (n > 0 ) && ( local.charAt(n - 1 ) == symb )) n++ ;
    else // not dup
    {
        switch( symb )
        {
            case 'A' : case 'E' : case 'I' : case 'O' : case 'U' :
                if( n == 0 )
                {
                    code.append(symb );mtsz++;
                }
                break ; // only use vowel if leading char
            case 'B' :
                if( (n > 0 ) &&
                    !(n + 1 == wdsz ) && // not MB at end of word
                    ( local.charAt(n - 1) == 'M'))
                    {
                        code.append(symb);
                    }
                else if (n==0)
                    {code.append(symb);}
                mtsz++ ;
                break ;
            case 'C' : // lots of C special cases
                // discard if SCI, SCE or SCY
                if( ( n > 0 ) &&
                    ( local.charAt(n-1) == 'S' ) &&
                    ( n + 1 < wdsz ) &&
                    ( FRONTV.indexOf( local.charAt(n + 1)) >= 0 )){ break ;}
                tmpS = local.toString();
                if( tmpS.indexOf("CIA", n ) == n ) // "CIA" -> X
                {
                    code.append('X' ); mtsz++; break ;
                }
                if( ( n + 1 < wdsz ) &&
                    (FRONTV.indexOf( local.charAt(n+1) )>= 0 ))
                {
                    code.append('S');mtsz++; break ; // CI,CE,CY -> S
                }
                if(( n > 0) &&
                    ( tmpS.indexOf("SCH",n-1 )== n-1 ))     // SCH->sk
                {
                    code.append('K') ; mtsz++;break ;
                }
                if( tmpS.indexOf("CH", n ) == n ) // detect CH
            {
                if((n == 0 ) &&
                     (wdsz >= 3 ) &&    // CH consonant -> K consonant
                     (VOWELS.indexOf( local.charAt( 2) ) < 0 ))
                {
                     code.append('K');
                }
                else
                {
                    code.append('X'); // CHvowel -> X
                }
                mtsz++;
            }
            else
            {
                code.append('K' );mtsz++;
            }
            break ;
        case 'D' :
            if(( n + 2 < wdsz )&&  // DGE DGI DGY -> J
             ( local.charAt(n+1) == 'G' )&&
             (FRONTV.indexOf( local.charAt(n+2) )>= 0))
            {
                code.append('J' ); n += 2 ;
            }
            else
            {
                code.append( 'T' );
            }
            mtsz++;
            break ;
        case 'G' : // GH silent at end or before consonant
            if(( n + 2 == wdsz )&&
             (local.charAt(n+1) == 'H' )) break ;
            if(( n + 2 < wdsz ) &&
             (local.charAt(n+1) == 'H' )&&
             (VOWELS.indexOf( local.charAt(n+2)) < 0 )) break ;
            tmpS = local.toString();
            if((n > 0) &&
                ( tmpS.indexOf("GN", n ) == n)||
                ( tmpS.indexOf("GNED",n) == n )) break ; // silent G
            if(( n > 0 ) &&
                (local.charAt(n-1) == 'G')) hard = true ;
            else hard = false ;
            if((n+1 < wdsz) &&
                (FRONTV.indexOf( local.charAt(n+1) ) >= 0 )&&
                (!hard) ) code.append( 'J' );
            else code.append('K');
            mtsz++;
            break ;
```

```
case 'H':
   if( n + 1 == wdsz ) break ; // terminal H
   if((n > 0) &&
    (VARSON.indexOf( local.charAt(n-1)) >= 0)) break ;
   if( VOWELS.indexOf( local.charAt(n+1)) >=0 )
   {
      code.append('H') ; mtsz++;// Hvowel
   }
   break;
case 'F': case 'J' : case 'L' :
case 'M': case 'N' : case 'R' :
   code.append( symb ); mtsz++; break ;
case 'K' :
   if( n > 0 ) // not initial
   {
      if( local.charAt( n -1) != 'C' )
      {
          code.append(symb );
      }
   }
   else   code.append( symb ); // initial K
   mtsz++ ;
   break ;
case 'P' :
   if((n + 1 < wdsz) &&  // PH -> F
    (local.charAt( n+1) == 'H'))code.append('F');
   else code.append( symb );
   mtsz++;
   break ;
case 'Q' :
   code.append('K' );mtsz++; break ;
case 'S' :
   tmpS = local.toString();
   if((tmpS.indexOf("SH", n )== n) ||
      (tmpS.indexOf("SIO",n )== n) ||
      (tmpS.indexOf("SIA",n )== n)) code.append('X');
   else code.append( 'S' );
   mtsz++ ;
   break ;
case 'T' :
   tmpS = local.toString(); // TIA TIO -> X
   if    ((tmpS.indexOf("TIA",n )== n)||
       (tmpS.indexOf("TIO",n )== n) )
   {
      code.append('X'); mtsz++; break;
   }
   if( tmpS.indexOf("TCH",n )==n) break;
   // substitute numeral 0 for TH (resembles theta after all)
   if( tmpS.indexOf("TH", n )==n) code.append('0');
   else code.append( 'T' );
```

```
                    mtsz++ ;
                    break ;
                case 'V' :
                    code.append('F'); mtsz++;break ;
                case 'W' : case 'Y' : // silent if not followed by vowel
                    if((n+1 < wdsz) &&
                        (VOWELS.indexOf( local.charAt(n+1))>=0))
                    {
                        code.append( symb );mtsz++;
                    }
                    break ;
                case 'X' :
                    code.append('K'); code.append('S');mtsz += 2;
                    break ;
                case 'Z' :
                    code.append('S'); mtsz++; break ;
            } // end switch
            n++ ;
        } // end else from symb != 'C'
        //if( mtsz > 4 )code.setLength( 4);
    } //end while
    return code.toString();
} // end static method code()
 */
protected static boolean slavoGermanic(String arg)
{
    if ((arg.indexOf('W') > -1)
        || (arg.indexOf('K') > -1)
        || (arg.indexOf("CZ") > -1)
        || (arg.indexOf("WITZ") > -1))
        {return true;}
    else
        {return false;}
}
protected static boolean isVowel(int at,String arg)
{
    if ((at < 0) || (at >= (arg.length()-5)))   {return false;}
    return VOWELS.indexOf(arg.charAt(at)) > -1;
}
protected static boolean StringAt(int start,String arg, String test)
{
    String[] work = {test};
    return StringAt(start,arg,work);
}
protected static boolean StringAt(int start,String arg, String[] tests)
{
    if (start < 0) return false;
    for (int i=0;i<tests.length;i++)
    {
        if (arg.startsWith(tests[i],start))
```

```
                {return true;}
            }
            return false;
        }
        public static String code( String arg)
        {
            return code (arg, 0);
        }
        public static String code( String arg , int code)
        {
            if (code!=0) {code=1;}
            return altCode(arg)[code];
        }
        public static void altCode(String arg, String[] code, String[] code_alt)
        {
            String[] retValues = altCode(arg);
            code[0]     = retValues[0];
            code_alt[0] = retValues[1];
        }
        public static Vector altCodeV(String arg)
        {
            Vector returnV = new Vector();
            String[] retValues = altCode(arg);
            returnV.add(0,retValues[0]);
            returnV.add(1,retValues[1]);
            return returnV;
        }
        public static String[] altCode(String arg)
        {
            int current = 0;
            int len = arg.length();
            int last = len - 1;//zero based index
            //String          primary, secondary;
            String[] retValues = new String[2];
            //Vector returnV = new Vector();
            for (int i=0;i<retValues.length;i++){retValues[i] = "";}
            if (len < 1)
            {
//              returnV.add(0,retValues[0]);
//              returnV.add(1,retValues[1]);
                return retValues;
            }
            boolean alternate = false;
            arg = arg.toUpperCase();
            //pad the original string so that we can index beyond the edge of the world
            arg += "     ";
            //skip these when at start of word: "GN", "KN", "PN", "WR", "PS"
            if (StringAt(0, arg,WORD_START_SKIP_1))
                {current += 1;}
            //Initial 'X' is pronounced 'Z' e.g. 'Xavier'
```

```
if (arg.charAt(0) == 'X')
{
   retValues[0] += "S";//'Z' maps to 'S'
   retValues[1] += "S";//'Z' maps to 'S'
   current += 1;
}
///////////main loop////////////////////////
while(current < len)
{
   switch(arg.charAt(current))
   {
      case 'A':
      case 'E':
      case 'I':
      case 'O':
      case 'U':
      case 'Y':
         if (current == 0)
         {
            //all init vowels now map to 'A'
            retValues[0] += "A";
            retValues[1] += "A";
         }
         current +=1;
         break;
      case 'B':
         //"-mb", e.g", "dumb", already skipped over...
         retValues[0] += "P";
         retValues[1] += "P";
         if (arg.charAt(current + 1) == 'B')
            {current +=2;}
         else
            {current +=1;}
         break;
      case 'C': // ASCII 199 (Extended ASCII removed for PTO ePAVE acceptance)
         retValues[0] += "S";
         retValues[1] += "S";
         current += 1;
         break;
      case 'C':
            //various germanic
         if ( (current > 1)
             && !isVowel(current - 2,arg)
             && StringAt((current - 1),arg,"ACH")
             && ( (arg.charAt(current + 2) != 'I')
                && ((arg.charAt(current + 2) != 'E')
                   || StringAt((current - 2), arg, C_GERMANIC)
                   )
                )
             )
```

```
{
   retValues[0] += "K";
   retValues[1] += "K";
   current +=2;
   break;
}
//special case 'caesar'
if ((current == 0) && StringAt(current, arg, "CAESAR"))
{
   retValues[0] += "S";
   retValues[1] += "S";
   current +=2;
   break;
}
//italian 'chianti'
if (StringAt(current, arg, "CHIA"))
{
      retValues[0] += "K";
      retValues[1] += "K";
      current +=2;
      break;
}
if (StringAt(current, arg, "CH"))
{
   //find 'michael'
   if ((current > 0) && StringAt(current, arg, "CHAE"))
   {
         retValues[0] += "K";
         alternate = true;
         retValues[1] += "X";
         current +=2;
         break;
   }
   //greek roots e.g. 'chemistry', 'chorus'
   if ((current == 0)
      && StringAt((current + 1), arg,C_GREEK)
      && !StringAt(0, arg, "CHORE"))
   {
      retValues[0] += "K";
      retValues[1] += "K";
      current +=2;
      break;
   }
   //germanic, greek, or otherwise 'ch' for 'kh' sound
   if ((StringAt(0, arg,GERMANIC))
      // 'architect but not 'arch', 'orchestra', 'orchid'
      || StringAt((current - 2), arg, C_RCH)
      || StringAt((current + 2), arg, C_TS)
      || ((StringAt((current - 1), arg, C_AOUE) || (current == 0))
      //e.g., 'wachtler', 'wechsler', but not 'tichner'
```

```
        && StringAt((current + 2), arg, C_LRNMBHFVW)))
    {
        retValues[0] += "K";
        retValues[1] += "K";
    }
    else
    {
        if (current > 0)
        {
            if (StringAt(0, arg, "MC"))
            {
                //e.g., "McHugh"
                retValues[0] += "K";
                retValues[1] += "K";
            }
            else
            {
                retValues[0] += "X";
                alternate = true;
                retValues[1] += "K";
            }
        }
        else
        {
            retValues[0] += "X";
            retValues[1] += "X";
        }
    }
    current +=2;
    break;
}
//e.g, 'czerny'
if (StringAt(current, arg, "CZ") && !StringAt((current - 2), arg, "WICZ"))
{
    retValues[0] += "S";
    alternate = true;
    retValues[1] += "X";
    current += 2;
    break;
}
//e.g., 'focaccia'
if (StringAt((current + 1), arg, "CIA"))
{
    retValues[0] += "X";
    retValues[1] += "X";
    current += 3;
    break;
}
//double 'C', but not if e.g. 'McClellan'
if (StringAt(current, arg, "CC") && !((current == 1) && (arg.charAt(0) == 'M')))
```

```
{
    //'bellocchio' but not 'bacchus'
    if (StringAt((current + 2), arg, C_IEH) && !StringAt((current + 2), arg, "HU"))
    {
        //'accident', 'accede' 'succeed'
        if (((current == 1) && (arg.charAt(current - 1) == 'A'))
            || StringAt((current - 1), arg,C_UCC))
        {
            retValues[0] += "KS";
            retValues[1] += "KS";
        }
        //'bacci', 'bertucci', other italian
        else
        {
            retValues[0] += "X";
            retValues[1] += "X";
        }
        current += 3;
        break;
    }
    else
    {//Pierce's rule
        retValues[0] += "K";
        retValues[1] += "K";
        current += 2;
        break;
    }
}
if (StringAt(current, arg, C_CKCGCQ))
{
        retValues[0] += "K";
        retValues[1] += "K";
        current += 2;
        break;
}
if (StringAt(current, arg, C_CICECY))
{
    //italian vs. english
    if (StringAt(current, arg, C_CIOCIECIA))
    {
        retValues[0] += "S";
        alternate = true;
        retValues[1] += "X";
    }
    else
    {
        retValues[0] += "S";
        retValues[1] += "S";
    }
    current += 2;
```

```
            break;
        }
        //else
        retValues[0] += "K";
        retValues[1] += "K";
        //name sent in 'mac caffrey', 'mac gregor
        if (StringAt((current + 1), arg,C_CQG))
            {current += 3;}
        else
        {
            if (StringAt((current + 1), arg,C_CKQ)
                    && !StringAt((current + 1), arg,C_CECI))
                {current += 2;}
            else
                {current += 1;}
        }
        break;
    case 'D':
        if (StringAt(current, arg, "DG"))
        {
            if (StringAt((current + 2), arg,D_IEY))
            {
                //e.g. 'edge'
                retValues[0] += "J";
                retValues[1] += "J";
                current += 3;
                break;
            }else{
                //e.g. 'edgar'
                retValues[0] += "TK";
                retValues[1] += "TK";
                current += 2;
                break;
            }
        }
        //arabic DH->D
        //skip double D equivalent DHDH
        if (StringAt(current, arg, "DH"))
        {
            if (!StringAt(current-2, arg, "DH") )
            {
                retValues[0] += "T";
                retValues[1] += "T";
            }
            current += 2;
            break;
        }
        if (StringAt(current, arg, D_DTDD))
        {
            retValues[0] += "T";
```

```
                retValues[1] += "T";
                current += 2;
                break;
            }
            //else
            retValues[0] += "T";
            retValues[1] += "T";
            current += 1;
            break;
        case 'F':
            if (arg.charAt(current + 1) == 'F')
                    current += 2;
            else
                    current += 1;
            retValues[0] += "F";
            retValues[1] += "F";
            break;
        case 'G':
            if (arg.charAt(current + 1) == 'H')
            {
                if ((current > 0) && !isVowel(current - 1,arg))
                {
                    retValues[0] += "K";
                    retValues[1] += "K";
                    current += 2;
                    break;
                }
                if (current < 3)
                {
                //'ghislane', ghiradelli
                    if (current == 0)
                    {
                        if (arg.charAt(current + 2) == 'I')
                        {
                            retValues[0] += "J";
                            retValues[1] += "J";
                        }
                        else
                        {
                            retValues[0] += "K";
                            retValues[1] += "K";
                        }
                        current += 2;
                        break;
                    }
                }
                //Parker's rule (with some further refinements) - e.g., 'hugh'
                if (((current > 1) && StringAt((current - 2), arg,G_BHD) )
                    //e.g., 'bough'
                    || ((current > 2) && StringAt((current - 3), arg, G_BHD) )
            //e.g., 'broughton'
            || ((current > 3) && StringAt((current - 4), arg, G_BHD) ) )
        {
            current += 2;
            break;
        }
        else
        {
            //e.g., 'laugh', 'McLaughlin', 'cough', 'gough', 'rough', 'tough'
            if ((current > 2)
                && (arg.charAt(current - 1) == 'U')
                && StringAt((current - 3), arg, G_CGLRT) )
            {
                retValues[0] += "F";
                retValues[1] += "F";
            }
            else
            {
                if ((current > 0) && arg.charAt(current - 1) != 'I')
                {
                    retValues[0] += "K";
                    retValues[1] += "K";
                }
            }
            current += 2;
            break;
        }
    }
    if (arg.charAt(current + 1) == 'N')
    {
        if ((current == 1) && isVowel(0,arg) && !slavoGermanic(arg))
        {
                retValues[0] += "KN";
                alternate = true;
                retValues[1] += "N";
        }
        else
        {
            //not e.g. 'cagney'
            if (!StringAt((current + 2), arg, "EY")
                && (arg.charAt(current + 1) != 'Y') && !slavoGermanic(arg))
            {
                retValues[0] += "N";
                alternate = true;
                retValues[1] += "KN";
            }
            else
            {
                retValues[0] += "KN";
                retValues[1] += "KN";
        }
    }
    current += 2;
    break;
}
//'tagliaro'
if (StringAt((current + 1), arg, "LI") && !slavoGermanic(arg))
{
    retValues[0] += "KL";
    alternate = true;
    retValues[1] += "L";
    current += 2;
    break;
}
//-ges-,-gep-,-gel-, -gie- at beginning
// "GES", "GEP", "GEB", "GEL", "GEY", "GIB", "GIL", "GIN", "GIE", "GEI", "GER"
if ((current == 0)
    && ((arg.charAt(current + 1) == 'Y')
    || StringAt((current + 1), arg, G_GES_GEP)) )
{
    retValues[0] += "K";
    alternate = true;
    retValues[1] += "J";
    current += 2;
    break;
}
// -ger-,  -gy-
// "DANGER", "RANGER", "MANGER"
if ((StringAt((current + 1), arg, "ER") || (arg.charAt(current + 1) == 'Y'))
    && !StringAt(0, arg, G_GER_GY)
    && !StringAt((current - 1), arg, G_EI)
    && !StringAt((current - 1), arg, G_RGYOGY) )
{
    retValues[0] += "K";
    alternate = true;
    retValues[1] += "J";
    current += 2;
    break;
}
// italian e.g, 'biaggi'
if (StringAt((current + 1), arg, G_EIY)
    || StringAt((current - 1), arg, G_ITALIAN))
{
    //obvious germanic
    if (StringAt(0, arg, GERMANIC)
        || StringAt((current + 1), arg, "ET"))
    {
        retValues[0] += "K";
        retValues[1] += "K";
    }
```

```
                    else
                    {
                        //always soft if french ending
                        if (StringAt((current + 1), arg, "IER "))
                        {
                            retValues[0] += "J";
                            retValues[1] += "J";
                        }
                        else
                        {
                            retValues[0] += "J";
                            alternate = true;
                            retValues[1] += "K";
                        }
                    }
                    current += 2;
                    break;
                }
                if (arg.charAt(current + 1) == 'G')
                    {current += 2;}
                else
                    {current += 1;}
                retValues[0] += "K";
                retValues[1] += "K";
                break;
            case 'H':
                //only keep if first & before vowel or btw. 2 vowels
//              System.out.println("I am here");
//              System.out.println("(current == 0):"+(current == 0));
//              System.out.println("isVowel(current - 1,arg)"+isVowel(current - 1,arg));
//              System.out.println("VOWELS.indexOf(arg.charAt(current + 1))"+VOWELS.indexOf(arg.charAt(current +
1)));
//              System.out.println("isVowel(current + 1,arg)"+isVowel(current + 1,arg));
//              System.out.println("arg"+arg);
                if (((current == 0) || isVowel(current - 1,arg))
                    && isVowel(current + 1,arg))
                {
//                  if (arg.equalsIgnoreCase("Hadlee"))
//                  {
//                      System.out.println("I am here") ;
//                  }
                    retValues[0] += "H";
                    retValues[1] += "H";
                    current += 2;
                }
                else //also takes care of 'HH'
                {
                    current += 1;
                }
                break;
```

```
case 'J':
    //obvious spanish, 'jose', 'san jacinto'
    if (StringAt(current, arg, "JOSE") || StringAt(0, arg, "SAN ") )
    {
        if (((current == 0) && (arg.charAt(current + 4) == ' '))
            || StringAt(0, arg, "SAN ") )
        {
            retValues[0] += "H";
            retValues[1] += "H";
        }
        else
        {
                retValues[0] += "J";
                alternate = true;
                retValues[1] += "H";
        }
        current +=1;
        break;
    }
    if ((current == 0) && !StringAt(current, arg, "JOSE"))
    {
        retValues[0] += "J";
        alternate = true;
        retValues[1] += "A";//Yankelovich/Jankelowicz
    }
    else
    {
        //spanish pron. of e.g. 'bajador'
        if (isVowel(current - 1,arg)
            && !slavoGermanic(arg)
            && ((arg.charAt(current + 1) == 'A') || (arg.charAt(current + 1) == 'O')))
        {
            retValues[0] += "J";
            alternate = true;
            retValues[1] += "H";
        }
        else
        {
            if (current == last)
            {
                retValues[0] += "J";
                alternate = true;
            }
            else
            {
                if (!StringAt((current + 1), arg, J_LTKSNMBZ)
                    && !StringAt((current - 1), arg, J_SKL))
                {
                    retValues[0] += "J";
                    retValues[1] += "J";
                }
            }
        }
    }
    if (arg.charAt(current + 1) == 'J')//it could happen!
        {current += 2;}
    else
        {current += 1;}
    break;
case 'K':
    //this is to add arabic KH -> H equivalence
    if ((current == 0)
        && arg.charAt(current + 1) == 'H'
        && isVowel((current + 2),arg))
    {
        current += 3;
        retValues[0] += "H";
        retValues[1] += "H";
        break;
    }
    if (arg.charAt(current + 1) == 'K')
            {current += 2;}
    else
            {current += 1;}
    retValues[0] += "K";
    retValues[1] += "K";
    break;
case 'L':
    if (arg.charAt(current + 1) == 'L')
    {
        //spanish e.g. 'cabrillo', 'gallegos'
        if (((current == (len - 3))
            && StringAt((current - 1), arg, L_SPANISH))
            || ((StringAt((last - 1), arg, L_ASOS) || StringAt(last, arg, L_AO))
            && StringAt((current - 1), arg, "ALLE")) )
        {
            retValues[0] += "L";
            alternate = true;
            current += 2;
            break;
        }
        current += 2;
    }
    else
        {current += 1;}
    retValues[0] += "L";
    retValues[1] += "L";
    break;
case 'M':
    if ((StringAt((current - 1), arg, "UMB")
        && (((current + 1) == last) || StringAt((current + 2), arg, "ER")))
        //'dumb','thumb'
        ||  (arg.charAt(current + 1) == 'M') )
    {
        current += 2;
    }
    else
    {
        current += 1;
    }
    retValues[0] += "M";
    retValues[1] += "M";
    break;
case 'N':
    if (arg.charAt(current + 1) == 'N')
        {current += 2;}
    else
        {current += 1;}
    retValues[0] += "N";
    retValues[1] += "N";
    break;
case 'N':  // ASCII 209 (Extended ASCII removed for PTO ePAVE acceptance)
    current += 1;
    retValues[0] += "N";
    retValues[1] += "N";
    break;
case 'P':
    if (arg.charAt(current + 1) == 'H')
    {
        retValues[0] += "F";
        retValues[1] += "F";
        current += 2;
        break;
    }
    //also account for "campbell", "raspberry"
    if (StringAt((current + 1), arg, P_PB))
        {current += 2;}
    else
        {current += 1;}
    retValues[0] += "P";
    retValues[1] += "P";
    break;
case 'Q':
    if (arg.charAt(current + 1) == 'Q')
        {current += 2;}
    else
        {current += 1;}
    retValues[0] += "K";
    retValues[1] += "K";
    break;
```

```
case 'R':
    //french e.g. 'rogier', but exclude 'hochmeier'
    if ((current == last)
        && !slavoGermanic(arg)
        && StringAt((current - 2), arg, "IE")
        && !StringAt((current - 4), arg, R_MEMA))
    {
        alternate = true;
        retValues[1] += "R";
    }
    else
    {
        retValues[0] += "R";
        retValues[1] += "R";
    }
    if (arg.charAt(current + 1) == 'R')
        {current += 2;}
    else
        {current += 1;}
    break;
case 'S':
    //special cases 'island', 'isle', 'carlisle', 'carlysle'
    if (StringAt((current - 1), arg, S_ISLYSL))
    {
        current += 1;
        break;
    }
    //special case 'sugar-'
    if ((current == 0) && StringAt(current, arg, "SUGAR"))
    {
        retValues[0] += "X";
        alternate = true;
        retValues[1] += "S";
        current += 1;
        break;
    }
    if (StringAt(current, arg, "SH"))
    {
        //germanic
        if (StringAt((current + 1), arg, S_GERMANIC))
        {
            retValues[0] += "S";
            retValues[1] += "S";
        }
        else
        {
            retValues[0] += "X";
            retValues[1] += "X";
        }
        current += 2;
        break;
    }
    //italian & armenian
    if (StringAt(current, arg, S_ITALIAN))
    {
        if (!slavoGermanic(arg))
        {
            retValues[0] += "S";
            alternate = true;
            retValues[1] += "X";
        }
        else
        {
            retValues[0] += "S";
            retValues[1] += "S";
        }
        current += 3;
        break;
    }
    //german & anglicisations, e.g. 'smith' match 'schmidt', 'snider' match 'schneider'
    //also, -sz- in slavic language altho in hungarian it is pronounced 's'
    if (((current == 0)
        && StringAt((current + 1), arg, S_MNLW))
        || StringAt((current + 1), arg, "Z"))
    {
        retValues[0] += "S";
        alternate = true;
        retValues[1] += "X";
        if (StringAt((current + 1), arg, "Z"))
            {current += 2;}
        else
            {current += 1;}
        break;
    }
    if (StringAt(current, arg, "SC"))
    {
        //Schlesinger's rule
        if (arg.charAt(current + 2) == 'H')
        {
            //dutch origin, e.g. 'school', 'schooner'
            if (StringAt((current + 3), arg, S_DUTCH))
            {
                //'schermerhorn', 'schenker'
                if (StringAt((current + 3), arg, S_DUTCH_2))
                {
                    retValues[0] += "X";
                    alternate = true;
                    retValues[1] += "SK";
                }
                else
```

```
                            {
                                retValues[0] += "SK";
                                retValues[1] += "SK";
                            }
                            current += 3;
                            break;
                        }
                        else
                        {
                            if ((current == 0) && !isVowel(3,arg) && (arg.charAt(3) != 'W'))
                            {
                                retValues[0] += "X";
                                alternate = true;
                                retValues[1] += "S";
                            }
                            else
                            {
                                retValues[0] += "X";
                                retValues[1] += "X";
                            }
                            current += 3;
                            break;
                        }
                    }
                    if (StringAt((current + 2), arg, G_EIY))
                    {
                        retValues[0] += "S";
                        retValues[1] += "S";
                        current += 3;
                        break;
                    }
                    //else
                    retValues[0] += "SK";
                    retValues[1] += "SK";
                    current += 3;
                    break;
                }
                //french e.g. 'resnais', 'artois'
                if ((current == last) && StringAt((current - 2), arg, S_FRENCH))
                {
                    alternate = true;
                    retValues[1] += "S";
                }
//stripped ending "S" for ignoring plurals, but not for alternates :-( nvr
else if ((current == last) && !StringAt((current - 1), arg, VOWELS))
                {
                    alternate = true;
                    retValues[1] += "S";
                }
                else
```

```
        {
            retValues[0] += "S";
            retValues[1] += "S";
        }
        if (StringAt((current + 1), arg, S_SZ))
            {current += 2;}
        else
            {current += 1;}
        break;
    case 'T':
        if (StringAt(current, arg, "TION"))
        {
            retValues[0] += "X";
            retValues[1] += "X";
            current += 3;
            break;
        }
        if (StringAt(current, arg, T_TIATCH))
        {
            retValues[0] += "X";
            retValues[1] += "X";
            current += 3;
            break;
        }
        if (StringAt(current, arg, "TH")
            || StringAt(current, arg, "TTH"))
        {
            //special case 'thomas', 'thames' or germanic
            if (StringAt((current + 2), arg, T_OMAM)
                || StringAt(0, arg, GERMANIC))
            {
                retValues[0] += "T";
                retValues[1] += "T";
            }
            else
            {
                retValues[0] += "0";
                alternate = true;
                retValues[1] += "T";
            }
            current += 2;
            break;
        }
        if (StringAt((current + 1), arg,T_TD))
            {current += 2;}
        else
            {current += 1;}
        retValues[0] += "T";
        retValues[1] += "T";
        break;
```

```
case 'V':
   if (arg.charAt(current + 1) == 'V')
      {current += 2;}
   else
      {current += 1;}
   retValues[0] += "F";
   retValues[1] += "F";
   break;
case 'W':
   //can also be in middle of word
   if (StringAt(current, arg, "WR"))
   {
      retValues[0] += "R";
      retValues[1] += "R";
      current += 2;
      break;
   }
   if ((current == 0)
      && (isVowel(current + 1,arg) || StringAt(current, arg, "WH")))
   {
      //Wasserman should match Vasserman
      if (isVowel(current + 1,arg))
      {
         retValues[0] += "A";
         alternate = true;
         retValues[1] += "F";
      }
      else
      {
         //need Uomo to match Womo
         retValues[0] += "A";
         retValues[1] += "A";
      }
   }
   //Arnow should match Arnoff
   if (((current == last) && isVowel(current - 1,arg))
      || StringAt((current - 1), arg, T_SLAVIC)
      || StringAt(0, arg, "SCH"))
   {
      alternate = true;
      retValues[1] += "F";
      current +=1;
      break;
   }
   //polish e.g. 'filipowicz'
   if (StringAt(current, arg, T_POLISH))
   {
      retValues[0] += "TS";
      alternate = true;
      retValues[1] += "FX";
```

```
            current +=4;
            break;
        }
        //else skip it
        current +=1;
        break;
    case 'X':
        //french e.g. breaux
        if (!((current == last)
            && (StringAt((current - 3), arg, X_IAUEAU)
            || StringAt((current - 2), arg, X_AUOU))) )
        {
            retValues[0] += "KS";
            retValues[1] += "KS";
        }
        if (StringAt((current + 1), arg, X_CX))
            {current += 2;}
        else
            {current += 1;}
        break;
    case 'Z':
        //chinese pinyin e.g. 'zhao'
        if (arg.charAt(current + 1) == 'H')
        {
            retValues[0] += "J";
            retValues[1] += "J";
            current += 2;
            break;
        }
        else
        {
            if (StringAt((current + 1), arg, Z_ZOZIZA)
                || (slavoGermanic(arg) && ((current > 0) && arg.charAt(current - 1) != 'T')))
            {
                retValues[0] += "S";
                alternate = true;
                retValues[1] += "TS";
            }
            else
            {
                retValues[0] += "S";
                retValues[1] += "S";
            }
        }
        if (arg.charAt(current + 1) == 'Z')
            {current += 2;}
        else
            {current += 1;}
        break;
    default:
```

```
                current += 1;
            }
        }
//      returnV.add(0,retValues[0]);
//      returnV.add(1,retValues[1]);
        return retValues;
        //metaph  = primary;
        //only give back 4 char metaph
        //if (metaph.GetLength() > 4)
        //        metaph.SetAt(4,'\0');
        //if (alternate)
        //{
            //metaph2 = secondary;
                //if (metaph2.GetLength() > 4)
                //        metaph2.SetAt(4,'\0');
        //}
    }
    /*
    */
    public static void main(String[] args)
    {
        if (args[0].equalsIgnoreCase("-f") && args.length == 3)
        {
            String inputFile=args[1],outputFile=args[2];
            BufferedReader input;
            PrintWriter output;
            String currFilename="";
            try
            {
                currFilename = inputFile;
                input = new BufferedReader(new FileReader(currFilename));
                currFilename = outputFile;
                output = new PrintWriter(new FileWriter(currFilename));
                String buffer;
                currFilename = inputFile;
                buffer = input.readLine();
                while (buffer!=null)
                {
                    Vector work = Match.altCodeV(buffer);
                    currFilename = outputFile;
                    output.println(buffer+" "+work.get(0)+" "+work.get(1));
                    currFilename = inputFile;
                    buffer = input.readLine();
                }
                output.flush();
                output.close();
                input.close();
            }
            catch (IOException ex)
            {
                System.err.println("Error in file:"+currFilename+" Error:" +ex.getMessage() );
                ex.printStackTrace(System.err);
            }
        }
        else if (args[0].equalsIgnoreCase("-s") && args.length == 2)
        {
            String code = Match.code(args[1]);
            Vector altCode = Match.altCodeV(args[1]);
            System.out.println(args[1]+" "+code);
            System.out.println(args[1]+" "+(String)altCode.get(0)+" "+(String)altCode.get(1));
        }
        else if (args[0].equalsIgnoreCase("-s") && args.length == 3)
        {
            Match jw =new Match();
            String s1 = args[1];
            String s2 = args[2];
            double jwOne = jw.score(s1,s2);
            System.out.println("score 1("+s1.length()+") to 2("+s2.length()+")="+jwOne);
            double jwTwo = jw.score(s2,s1);
            System.out.println("score 2("+s2.length()+") to 1("+s1.length()+")="+jwTwo);
            String mpOne = jw.code(s1);
            System.out.println("code\t 1("+s1.length()+") ="+mpOne);
            String mpTwo = jw.code(s2);
            System.out.println("code\t 2("+s2.length()+") ="+mpTwo);
            Vector altmpOne = jw.altCodeV(s1);
            System.out.println("altCode\t 1("+s1.length()+") ="+(String)altmpOne.get(0)+" "+(String)altmpOne.get(1));
            Vector altmpTwo = jw.altCodeV(s2);
            System.out.println("altCode\t 2("+s2.length()+") ="+(String)altmpTwo.get(0)+" "+(String)altmpTwo.get(1));
            double jwThree = jw.score(mpOne,mpTwo);
            System.out.println("score meta 1("+mpOne.length()+") to 2("+mpTwo.length()+")="+jwThree);
            double jwFour = jw.score(mpTwo,mpOne);
            System.out.println("score meta 2("+mpTwo.length()+") to 1("+mpOne.length()+")="+jwFour);
        }
        else
        {
            System.out.println("Match\t:\t Syntax ");
            System.out.println("\t:\t -f <input file name> <output file name>");
            System.out.println("\t:\t or ");
            System.out.println("\t:\t -s <String to convert>");
            System.out.println("\t:\t or ");
            System.out.println("\t:\t -s <String to convert> <string to convert>");
        }
    }
}
// PatriotCommon.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
package com.sybase.patriotact.filter;
/*
*/
// IO Classes
```

```
import java.io.File;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
//Util Classes
import java.util.ArrayList;
import java.util.List;
import java.util.Hashtable;
import java.util.StringTokenizer;
import java.util.Vector;
import java.util.Enumeration;
//SQL Classes
import java.sql.PreparedStatement;
//Log4j
import org.apache.log4j.Logger;
import java.util.Hashtable;
import java.sql.Time;
//Patriot Act Util Classes
import com.sybase.patriotact.utils.DBConnection;
//Patriot Act Classes
import com.sybase.patriotact.filter.StopWords;
/** PatriotCommon serves as a code repository for functions & methods used throughout the PatriotAct solution. The
  * key methods that retrieve the potential matches, score the hits & check the cleared lists are all herein. Also
  * included are methods that build up the SQL strings needed for matching & methods to parse out unwanted words &
  * characters in strings.
  *
  * <br><br>The class also defines a number of static constants.
  *
  */
public class PatriotCommon {
 //static String onlyLettersMatchString = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
 static String wordMatchString = " ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";//Letters plus space character
 static String whiteSpaceString = " ";
 static String nuanceCharactersString = "/\\-,/\n";
 static StopWords ss = null;
 static Hashtable stopWords = null;
 static Hashtable scoreStopWords = null;
 static float SURNAMEWEIGHT = (float)1.2;   //Matches to surnames, (when found) are weighed more heavily.
 static float PRENOMWEIGHT = (float)0.8;  //Matches to forenames, (when found) are weighed less heavily.
 static int shortWordThreshold = Constants.SHORTWORDTHRESHOLD; //Words this length or shorted are ignored in
the matching. This usually gets overriden from the Filter.properties file.
 static float lowMatchThresholdValue = (float) 1.0; // A value of 1.0 is neutral. This usually gets overriden from the
Filter.properties file.
 //log4j
 private static Logger _log = Logger.getLogger(PatriotCommon.class);
 private static FilterPrefs _p = null; //This is the class that loads the filter preferences
 private static String sqlStringQuickMatchSelect   = "";
 private static String customKeywordList    = "_xyz1234xyz"; //default to a crazy string
 private static String exactCountryMatchSql   = "";
 private static String exactNameMatchSql   = "";
 private static String clearedListSql1     = "";
 private static String clearedListSql2     = "";
 private static String clearedListFuzzySearchSql  = "";
 private static String matchClearedListType   = Constants.FUZZY_SEARCH; //Default to fuzzy for cleared list searching
 private static Double clearedListScoreThreshold  = null;
 private static String intelligentSurnameMatching  = "";
 private static String goodSoundingMatchOverride  = "";
 private static double goodSoundingMatchOverrideThreshold = 76.00;//default
 private static double goodSoundingMatchOverrideThresholdFromScore = 760.00;//default
 private static double matchThresholdFromScore   = 800.00;//default
 static double scoreThreshold     = 80.0;//default
 private static boolean loadingSemaphore   = false;
 /*
 ** For long names, i.e. names with 'thresholdForMultipleSoundAlikes' or more words, then at least two words
 ** must sound alike for prematching to be considered a sucess, instead of the normal 1.
 */
 private static int thresholdForMultipleSoundAlikes = 5;
 //private static boolean inMemoryStatus = false;
 //private static Hashtable quickMatches = null;
 private LoadHash searchHash = null; //used for NightlyFilter searches
 public PatriotCommon () {
  if (_p == null) {
   System.out.println("PatriotCommon for release " + Constants.VERSIONNUM + "/01");
   _p = new FilterPrefs(); //This is the class that loads the filter preferences
   shortWordThreshold = new Integer(_p.getProperty("shortWordThreshold")).intValue();
   thresholdForMultipleSoundAlikes = new Integer(_p.getProperty("thresholdForMultipleSoundAlikes")).intValue();
   //Value between 0.0 & 1.0.
   lowMatchThresholdValue = new Float(_p.getProperty("oneWordScoreThreshold")).floatValue();
   sqlStringQuickMatchSelect = _p.getProperty("sqlStringQuickMatchSelect");
   exactCountryMatchSql = _p.getProperty("sqlExactCountryMatch");
   exactNameMatchSql = _p.getProperty("sqlExactNameMatch");
   clearedListSql1 = _p.getProperty("sqlClearedList1");
   clearedListSql2 = _p.getProperty("sqlClearedList2");
   clearedListFuzzySearchSql = _p.getProperty("sqlClearedListFuzzySearch");
   matchClearedListType = _p.getProperty("MatchClearedListType");
   customKeywordList = _p.getProperty("customKeywordList");
   //toggles for some optional stuff, intelligent surnames & good-sounding-matches
   intelligentSurnameMatching = _p.getProperty("intelligentSurnameMatching");
   goodSoundingMatchOverride = _p.getProperty("goodSoundingMatchOverride");
   //The cleared list has it's own threshold for fuzzy scoring as it might be held to a higher threshold than normal
matching
   clearedListScoreThreshold = new Double(_p.getProperty("ClearedListScoreThreshold"));
   //Good sounding matches match below the score threshold but are to be given a second opinion
   goodSoundingMatchOverrideThreshold = (new
Double(_p.getProperty("goodSoundingMatchOverrideThreshold"))).doubleValue();
   goodSoundingMatchOverrideThresholdFromScore = goodSoundingMatchOverrideThreshold*10;
   //get the score threshold
   scoreThreshold = (new Double(_p.getProperty(Constants.FILTER_SCORE_THRESHOLD))).doubleValue();
  _log.warn("Score matching threshold : " + scoreThreshold);
  //This is the score multiplied by ten as it arrives from the utils.score method
  matchThresholdFromScore = scoreThreshold * 10;
 }
}
public void setSearchHash(LoadHash searchHash) {
 this.searchHash = searchHash;
}
/*
** If the SuspectCreate program is run standalone this routine is used to provide a database connection for the
** stopwords class.
*/
public void setupConnectionForStopWords(java.sql.Connection conn) {
 _log.info("setupConnectionForStopWords");
 //Generate the stopwords arraylist from the static stopwords class
 if (ss == null) {
  ss = new StopWords();
 }
 if (stopWords == null) {
  stopWords = ss.readStopWords(conn);
 }
}
/** Parses the string removing all instances of words that exist in the PatStopWords table. These are words that
  * because of the commonality are to be excluded from the matching. Eaxmples could be words like 'COMPANY' or
  * 'BANK'.
  * @param String suspect
  * returns String suspect with all stop-words removed
  * @since 2.0
  * @version 2.1
  */
public String removeStopWords(String suspect) {
 _log.debug("parameters to method : removeStopWords :" + suspect + "<");
 suspect = suspect.trim();
 StringTokenizer st = new StringTokenizer(suspect);
 StringBuffer buf = new StringBuffer ();
 //Generate the stopwords arraylist from the static stopwords class
 if (ss == null) {
  ss = new StopWords();
 }
 if (stopWords == null) {
  stopWords = ss.readStopWords();
 }
 //For each word in the transaction name match it against all the stop-words
      while (st.hasMoreTokens()) {
  String word = st.nextToken();
  boolean stopWordBool = false;
  if (stopWords.containsKey(word)) {
   _log.info("Stopword " + word + " removed ");
   stopWordBool = true;
  }
  //If not a stop-word then add it to the output
  if (stopWordBool == false) {
   buf.append(word + " ");
  }
      }
 _log.info("return from method : removeStopWords :" + buf.toString().trim() + "<");
 return buf.toString().trim();
}
/** Parses the string removing all instances of words that exist in the PatStopWords table. These are words that
  * because of the commonality are to be excluded from the matching. Eaxmples could be words like 'COMPANY' or
  * 'BANK'.
  * @param String suspect
  * returns String suspect with all stop-words removed
  * @since 2.0
  * @version 2.1
  */
public String removeScoreStopWords(String suspect) {
 _log.debug("parameters to method : removeScoreStopWords :" + suspect + "<");
 suspect = suspect.trim();
 StringTokenizer st = new StringTokenizer(suspect);
 StringBuffer buf = new StringBuffer ();
 //Generate the stopwords arraylist from the static stopwords class
 if (ss == null) {
  ss = new StopWords();
 }
 if (scoreStopWords == null) {
  scoreStopWords = ss.readScoreMatchStopWords();
 }
 //For each word in the transaction name match it against all the stop-words
      while (st.hasMoreTokens()) {
  String word = st.nextToken();
  boolean stopWordBool = false;
  if (scoreStopWords.containsKey(word)) {
   _log.info("Stopword " + word + " removed ");
   stopWordBool = true;
  }
  //If not a stop-word then add it to the output
  if (stopWordBool == false) {
   buf.append(word + " ");
  }
      }
 _log.info("return from method : removeScoreStopWords :" + buf.toString().trim() + "<");
 return buf.toString().trim();
}
/** Parses the provided string removing words of (Constants.SHORTWORDTHRESHOLD) letters or less. This
  * value is overwritten by the 'shortWordThreshold' value in the 'Filter.properties' file.
  * This method is used frequently to remove short words from the matching process. Short
  * words are removed from the index & matching process entirely. The value of 'shortWordThreshold' should be
  * small enough so that important words are not lost but if it is too short then small words can clutter
  * up and slow the matching. A value of one, two or three is expected.
  * @param String that is to have short word removed from.
  * returns String with short words removed
  * @since 2.0
  */
public String removeShortWords(String suspect) {
   _log.debug("parameters to method : removeShortWords : " + suspect + "<");
 suspect = suspect.trim();
 StringTokenizer st = new StringTokenizer(suspect);
 StringBuffer buf = new StringBuffer();
 //Get the number of words to process
 int numberOfWordsInString = st.countTokens();
 //For each word in the string
 int numberOfWordsRemoved = 0;
 //For each word in the transaction name remove it if it's too short to be considered.
      while (st.hasMoreTokens()) {
  String word = st.nextToken();
  //Remove the short words
  if (word.length() <= shortWordThreshold) {
   _log.info("Short word : " + word + " found, it will be removed");
   numberOfWordsRemoved++;
   //Whoa - if we're going to end up with (almost) no words left in the string then stop
   //and just return the original string
   if (numberOfWordsRemoved >= numberOfWordsInString) {
    _log.info("No words left to test against. Lowering the shortword threshold.");
    shortWordThreshold = shortWordThreshold - 1;
    suspect = removeShortWords(suspect);
    shortWordThreshold = shortWordThreshold + 1;
    return suspect;
   }
  }
  else {
   buf.append(word + " ");
  }
 }
 _log.info("return from method : removeShortWords : " + buf.toString().trim() + "<");
 return buf.toString().trim();
}
/** Parses the provided string removing characters that are not in the range A to Z or the space charchter ' '.
  * The string will be shortened by the number of characters removed.
  * @param String that is to be parsed.
  * returns String with parsed characters removed.
  * @since 2.0
  */
public String removeNonAlphabeticCharacters(String inString) {
 _log.debug("parameter to method : removeNonAlphabetLetters : " + inString + "<");
 //Get rid of any whitespace that may be on the string
 inString = inString.trim();
 StringBuffer buf = new StringBuffer(inString);
 StringBuffer outBuf = new StringBuffer();
 /*
 ** Strip all non-alphabetic characters from a string.
 */
 //Remove non-letters
 int bufLength = buf.length();
 for (int i = 0; i < bufLength; i++) {
  char c = buf.charAt(i);
  if (wordMatchString.indexOf(c) >= 0 ) {
   outBuf.append(c);
  }
  /*
  ** Nuance Characters concept inspired by names like 'STATE/CAPITAL' where the words would
  ** otherwise be compressed into one.
  */
  else if (nuanceCharactersString.indexOf(c) >= 0 ) {
   outBuf.append(' ');
  }
 }
 _log.info("return from method : removeNonAlphabetLetters : " + outBuf.toString() + "<");
 return outBuf.toString();
}
/** Calculates the number of words in a string
  * @param String
  * returns int number of words in the string
  * @since 2.0
  */
public int calcNumberOfWordsInString (String inString) {
 //_log.debug("parameters to method : calcNumberOfWordsInString : " + inString + "<");
 StringTokenizer st = new StringTokenizer(inString);
 int wordCount = st.countTokens();
 _log.info("return from method : calcNumberOfWordsInString : " + wordCount);
 return wordCount;
}
/** This is the method that pulls together calls to other methods to process a string in accordance
  * to the prinicpals of Patriot matching. It calls other methods to remove stop words, non alphabetic
  * characters & short words.
  * @param String inString
  * @since 2.0
  */
public String removeUnwantedChars(String inString) {
 _log.debug("parameter to method : removeUnwantedChars : inString : " + inString + "<");
 //Remove stop words from the suspect string
 String suspectStringStoppedOut = removeStopWords(inString);
 //if there's nothing left then put the words back
 if (suspectStringStoppedOut == null || suspectStringStoppedOut.equals("")) {
  suspectStringStoppedOut = inString;
 }
 //Now remove non alphabetic letters
 suspectStringStoppedOut = removeNonAlphabeticCharacters(suspectStringStoppedOut);
 //Now remove short words
 suspectStringStoppedOut = removeShortWords(suspectStringStoppedOut);
```

```
  //At the finish just trim excess whitespace
  suspectStringStoppedOut = suspectStringStoppedOut.trim();
  _log.info("return from method : removeUnwantedChars : " + suspectStringStoppedOut);
  return suspectStringStoppedOut;
 }
 /** This is the main entry point into the search engine for online & nighly filter processing. It handles
   * the initial retrieving of potential hits, scoring those hits & finally checking if any matches
   * exist in the PatClrList table. Matches are returned if they score greater than the 'scorethreshold'
   * value in 'Filter.properties' or they score greater than 'goodSoundingMatchOverrideThreshold' value
   * in 'Filter.properties' and have all significant words sounding alike. This particular feature can be
   * turned off using the 'goodSoundingMatchOverride' value in 'Filter.properties'.
   * @param record Object. Currently the following objects can be passed <br>
   * {@link MoneyTransactionObject MoneyTransactionObject}<br>
   * {@link CustomerObject CustomerObject} <br>
   * {@link EmployeeInfoObject EmployeeInfoObject} <br>
   * {@link DbCustomerObject DbCustomerObject} <br>
   * {@link DbEmployeeObject DbEmployeeObject} <br>
   * {@link ClearedListObject ClearedListObject} <br>
   * @param matchingField String The field to match in the object, e.g. name, city, country.
   * @param sQLString String the SQL string to search the database with (Not used in 2.1.2). This will have been
created via calls
   * to one of the methods 'buildFuzzy<Name/Country>MatchSql' or 'buildExact<name/Country>MatchSql' or
   * 'buildFuzzyCustomMatchSql'.
   * @param suspectString String The strign to be matched against all the known bad guys in the database
   * @param matchType String Either Fuzzy or Exact matching is supported.
   * @param clearedListSqlString String a piece of Sql to tag onto the cleared list search that narrows down
   * the search of the cleared list to e.g. a particular account.
   * @param conn {@link java.sql.Connection Connection}
   * @return Array of {@link SuspectHitResult SuspectHitResult} containing all the hits for that one field
   * @since 2.0
   * @version 2.1
   */
 public ArrayList getResults(Object record, String matchingField, String sQLString, String suspectString, String
matchType, String clearedListSqlString, java.sql.Connection conn) throws PatriotSearchException {
  _log.info("parameters to  method : getResults : record : " + record + " matchingField : " + matchingField + " sQLString :
" + sQLString + " suspectString : " + suspectString + " matchType : " + matchType + " clearedListSqlString : " +
clearedListSqlString + " java.sql.Connection : " + conn );
  ArrayList suspectHitResults  = new ArrayList(); //Arrray for adding return values
  try {
   suspectHitResults = generateCodes(record, matchingField, suspectString, clearedListSqlString, conn);
   /*
   ** In previous version there was a seperate branch for exact & fuzzy matches. Now there is
   ** only a fuzzy match, and exact matches are treated as fuzzy matches that must match 100%.
   */
   //An exact match is treated as a fuzzy match until now.
   if (matchType.startsWith("exact")) {
    //Remove all matches less than 100%
    for (int i = 0; i < suspectHitResults.size(); i++) {
     SuspectHitResult shr = (SuspectHitResult) suspectHitResults.get(i);
     float score = new Float(shr.getMatchComment()).floatValue();
    if (score < 99.99) {
     _log.warn("Removing a less that perfect match for " + shr.getFieldValue());
     suspectHitResults.remove(i);
    }
   }
  }
 }
 catch (Exception e) {
  e.printStackTrace();
  throw new PatriotSearchException();
 }
 return suspectHitResults;
}
/** Build the SQL string needed to do an exact match on a 'name' field. The SQL string is sqlExactNameMatch
* from the Filter.properties file. That sql string's where clause can be altered, e.g. to
* exclude or include various types of suspects, e.g. countries. The designation of this as being a search
*  on a name field is arbitary. It can be used to search on any field, the criteria being that the SQL String
* will retrieve the range of values appropriate for that field.
* @param inString String The name string that is to be searched for in the database.
* @return String the SQL string ready for passing to the database.
* @since 2.0
*/
public String buildExactNameMatchSQL(String inString) {
 _log.info("parameters to method : buildExactNameMatchSQL : inString > " + inString + "<");
 StringBuffer sQLBuf = new StringBuffer();
 StringTokenizer st = new StringTokenizer(exactNameMatchSql);
 while (st.hasMoreTokens()) {
  String word = st.nextToken();
  if (word.equals(Constants.FILTER_SUBSTITUTION_PATTERN)) {
   sQLBuf.append("\"" + inString + "\"");
  }
  else {
   sQLBuf.append(word);
  }
  sQLBuf.append(" ");
 }
 _log.info("return from method : buildExactNameMatchSQL : " + sQLBuf.toString());
 return sQLBuf.toString();
}
/** Build the SQL string needed to do an exact match on a 'country' field. The SQL string is sqlExactCountryMatch
* from the Filter.properties file. That sql string's where clause can be altered, e.g. to
* exclude or include various types of suspects, e.g. names. The designation of this as being a search
* on a country field is arbitary. It can be used to search on any field, the criteria being that the SQL String
* will retrieve the range of values appropriate for that field.
* @param inString String The country string that is to be searched for in the database.
* @return String the SQL string ready for passing to the database.
* @since 2.0
*/
public String buildExactCountryMatchSQL(String inString) {
 _log.info("parameter to method : buildExactCountryMatchSQL : " + inString + "<");
```

```
  StringBuffer sQLBuf = new StringBuffer();
  StringTokenizer st = new StringTokenizer(exactCountryMatchSql);
  while (st.hasMoreTokens()) {
   String word = st.nextToken();
   if (word.equals(Constants.FILTER_SUBSTITUTION_PATTERN)) {
    sQLBuf.append("\"" + inString + "\"");
   }
   else {
    sQLBuf.append(word);
   }
   sQLBuf.append(" ");
  }
  _log.info("return from method : buildExactCountryMatchSQL : " + sQLBuf.toString());
  return sQLBuf.toString();
 }
 /** Check if the suspect is in the cleared list for the name, & list_type. The check can be either exact or
   * fuzzy depending on the 'MatchClearedListType' value in the
   * Filter.properties file. If the match is fuzzy then the 'ClearedListScoreThreshold' value in Filter.properties
   * is used to determine the score threshold for matches.
   *
   * @since 2.0
   * @version 2.1
   * @param matchString String The match returned from the database
   * @param suspectString String The string that was passed to the suspect matching for analysis
   * @param sqlString The extra sql needed for the where clause
   * @param entId long The unique identifier for the row in the PatMasterList table
   * @param altNum long The alt_num from the database. This is zero for rows from the master table and a positive
number
   * for rows from the alias table
   * @param originalWordInd long The original_word_ind from the PatQuickMatch table. This is zero for matchStrings
that have
   * no translations & a positive number for all matchStrings that have at least one word a translation.
   * @return null if the suspect is in the cleared list, otherwise it returns the partially populated
   * {@link SuspectHitResult SuspectHitResult}
   */
 public SuspectHitResult inClearedList (String matchString, String suspectString, String sqlString, long entId, long
altNum, long originalWordInd, java.sql.Connection connX) {
  _log.info("parameter to method : inClearedList : matchString : " + matchString + " : suspectString : " + suspectString +
" : sqlString : " + sqlString + " : entId : " + entId + " : altNum : " + altNum + " : originalWordInd : " + originalWordInd);
  String listType = "";
  String name = "";
  boolean clearedListHit = false;
  boolean isSecureList = false;
  java.sql.Connection conn = null;
  try {
   conn = DBConnection.getDBConnection();
   StringBuffer sQLBuf = new StringBuffer();
   StringTokenizer st = new StringTokenizer(clearedListSql1);
   while (st.hasMoreTokens()) {
    String word = st.nextToken();
 if (word.equals(Constants.FILTER_SUBSTITUTION_PATTERN)) {
  sQLBuf.append(entId);
 }
 else {
  sQLBuf.append(word);
 }
 sQLBuf.append(" ");
}
_log.info(sQLBuf.toString());
//First get the list_type for the match. Have to go back to the master table for this
java.sql.CallableStatement stm = conn.prepareCall(sQLBuf.toString());
       stm.execute();
       // Show result
       java.sql.ResultSet rss =stm.getResultSet();
//only expecting one row at most
       while ( rss.next() ) {
 listType = rss.getString("list_type").trim();
 name = rss.getString("name").trim();
 isSecureList = rss.getBoolean("is_secure");
 break;
}
if (listType == null || listType.equals("")) {
 _log.warn("Could not find a list type value for entity ID " + entId + ", name " + matchString);
 _log.warn("This test used the sqlClearedList1 value from the Filter.properties file.");
 _log.warn("A common cause of this test to fail is if the is_active flag is zero in PatMasterList.");
 _log.warn("The search engine will continue with it's cleared list checking without a list type value.");
 //return false;
}
_log.info("Searching this name in the cleared list with list type " + listType + "<");
sQLBuf = null;
sQLBuf = new StringBuffer();
StringTokenizer sqt = null;
if (matchClearedListType.equals(Constants.FUZZY_SEARCH)) {
 sqt = new StringTokenizer(clearedListFuzzySearchSql);
}
else {
 sqt = new StringTokenizer(clearedListSql2);
}
//Toggle the words for better cleared list matching, e.g. surname firstname; firstname surname
int hits = 0;
while (sqt.hasMoreTokens()) {
 String word = sqt.nextToken();
 if (word.equals(Constants.FILTER_SUBSTITUTION_PATTERN)) {
  if (hits == 0) {
   if (matchClearedListType.equals(Constants.FUZZY_SEARCH)) {
    //String workingSuspectString = compressSuspectName(suspectString);
    //Treat only the first 3 or less words as being important in the name
    String workingSuspectString = getOnlyFirstFewWords(suspectString, 3);
    workingSuspectString = removeNonAlphabeticCharacters(workingSuspectString);
    sQLBuf.append("\"" + addWildCards(workingSuspectString) + "\"");
```

```
     String reversedString = reverseWordsInString(workingSuspectString, 0);
     if (!reversedString.equals(workingSuspectString)) {
      sQLBuf.append(" or name like \"" + addWildCards(reversedString) + "\"");
      String reversedString2 = reverseWordsInString(workingSuspectString, 2);
      if (!reversedString2.equals(workingSuspectString)) {
       sQLBuf.append(" or name like \"" + addWildCards(reversedString2) + "\"");
      }
     }
    }
    else {
     sQLBuf.append("\"" + suspectString + "\"");
    }
   }
   else if (hits == 1) {
    sQLBuf.append(sqlString);
   }
   else if (hits == 2) {
    sQLBuf.append("\"" + listType + "\"");
   }
   hits++;
  }
  else {
   sQLBuf.append(word);
  }
  sQLBuf.append(" ");
 }
 _log.info(sQLBuf.toString());
 //The name must exist in the cleared list
 java.sql.CallableStatement stt = conn.prepareCall(sQLBuf.toString());
        stt.execute();
        // Show result
        java.sql.ResultSet rs =stt.getResultSet();
 if (matchClearedListType.equals(Constants.FUZZY_SEARCH)) {
         while ( rs.next() ) {
   String matchName = rs.getString("name").trim();
   //Score the names returned from the cleared list
   double matchscore = matchScores(matchName, suspectString, matchName, "", "", "", (float) 0.0, -1);
   if ((matchscore) >  clearedListScoreThreshold.doubleValue()) {
    clearedListHit = true; //at least one row so it's not a bad guy
    break; //only need one row
   }
  }
 }
 else {
         while ( rs.next() ) {
   clearedListHit = true; //at least one row so it's not a bad guy
   break; //only need one row
  }
 }
}
```

```
catch ( Exception e ) {
        System.out.println( "Suspect Match Critical ErrorX: " + e);
 _log.error("Suspect Match Critical ErrorXX " + e);
       e.printStackTrace();
    }
SuspectHitResult suspectHitResult = null;
if (clearedListHit == false) {
 try {
  suspectHitResult = new SuspectHitResultImpl ();
  /*
  ** If the match was on an alias then return the name from the master list with the alias name concatenated.
  ** Slight differences in the two versions of matching name can occur because matchString has had all the
  ** irrelevent characters removed while the name has not. Remove the non-alphabetic characters.
  */
  String keepName = name;
  name = removeNonAlphabeticCharacters(name.toUpperCase());
  if (!keepName.equalsIgnoreCase(matchString)) {
   if (originalWordInd != 1) {
    //True alias
    name = name + " (aka) " + matchString;
    suspectHitResult.setListFieldName("Alias");
   }
   else {
    //not a true alias, it's a hit on a translation of the original
    name = name + " (translation) " + matchString;
    suspectHitResult.setListFieldName("Name");
   }
  }
  else {
    suspectHitResult.setListFieldName("Name");
  }
  String aliasName = "";
  if (altNum > 0 && originalWordInd == Constants.TRANSLATION) {
   //Match was on a translation of an alias so get the original alias
   PreparedStatement pStatement = null;
   pStatement = conn.prepareStatement(_p.getProperty("sqlClearedListSqlName"));
   pStatement.setLong(1, entId);
   pStatement.setLong(2, altNum);
          pStatement.executeQuery();
          // Show result
          java.sql.ResultSet rsa =pStatement.getResultSet();
          while ( rsa.next() ) {//only one row expected
    aliasName = rsa.getString("alt_name").trim();
   }
  }
  suspectHitResult.setFieldName(keepName);
  //Tack on the original alias name if the hit was on a translation of the alias
  if (!aliasName.equals("")) {
   suspectHitResult.setFieldName(suspectHitResult.getFieldName() + " (translation of the alias) " + aliasName);
  }
```

```
   suspectHitResult.setFieldValue(suspectString);
   if (listType != null) {
    suspectHitResult.setListType(listType);
   }
   suspectHitResult.setListSecure(isSecureList);
   //If the hit was on a translation of an alias set the list field value to the original alias
   if (altNum > 0 && originalWordInd == Constants.TRANSLATION) {
    suspectHitResult.setListFieldValue(name + " of the alias " + aliasName);
   }
   else {
    //Set the name hit
    if (originalWordInd == Constants.TRANSLATION) {
     suspectHitResult.setListFieldValue(matchString + " (translation of) " + suspectHitResult.getFieldName());
    }
    else {
     suspectHitResult.setListFieldValue(matchString);
    }
   }
   suspectHitResult.setEntId(entId);
  }
  catch ( Exception e ) {
         System.out.println( "Suspect Match Critical ErrorY: " + e);
   _log.error("Suspect Match Critical ErrorYY " + e);
        e.printStackTrace();
     }
 }
 try {
  DBConnection.closeDBConnection(conn);
 }
 catch (Exception e) {
  e.printStackTrace();
 }
 _log.info("Return from method inClearedList : " + suspectHitResult);
 return suspectHitResult;
}
/** Check if all the words in two strings have at least one metaphone value the same.
* @param String suspectString The first of the two strings to check
* @param String matchingName The second string that is checked against the first string
* @return boolean true if all the words sound alike.
* @since 2.1
*/
public boolean allWordsSoundAlike(String suspectString, String matchingName) {
 _log.info("soundOutWords : " + suspectString + " matchingName " + matchingName);
 int numberSimilarSoundingWords = 0;
 boolean allWordsSoundAlike = false;
 int numberOfWordsInSuspectString = calcNumberOfWordsInString(suspectString);
 int numberOfWordsInMatchingString = calcNumberOfWordsInString(matchingName);
 Integer[] matchedWords = new Integer[numberOfWordsInMatchingString];
 StringTokenizer snm = new StringTokenizer(suspectString);
 String lastMatchingWord = "";
```

```
  while (snm.hasMoreTokens()) {
   String suspectWord = snm.nextToken();
   _log.debug("suspectWord : " + suspectWord);
   Vector suspectCodes = com.sybase.patriotact.utils.Match.altCodeV(suspectWord);
   String suspectCode = (String)suspectCodes.elementAt(0);
   String suspectAltCode = (String)suspectCodes.elementAt(1);
   StringTokenizer mnm = new StringTokenizer(matchingName);
   int matchingWordNumber = 0;
   while (mnm.hasMoreTokens()) {
    String matchingWord = mnm.nextToken();
    _log.debug("matchingWord : " + matchingWord);
    Vector matchingCodes = com.sybase.patriotact.utils.Match.altCodeV(matchingWord);
    String matchingCode = (String)matchingCodes.elementAt(0);
    String matchingAltCode = (String)matchingCodes.elementAt(1);
    if (suspectCode.equals(matchingCode) || suspectCode.equals(matchingAltCode) ||
     suspectAltCode.equals(matchingCode) || suspectAltCode.equals(matchingAltCode)) {
     /*
     ** Sometimes two or more words in a suspect string might have the same
     ** metaphone as a word in the matching name. Only allow the matching word
     ** to be considered once.
     */
     boolean alreadyMatched = false;
     for (int i = 0; i < numberSimilarSoundingWords; i++) {
      //if already matched to this word
      if (matchingWordNumber == matchedWords[i].intValue()) {
       alreadyMatched = true;
      }
     }
     if (alreadyMatched == true) {
      break;
     }
     _log.info("found two words that sound alike " + suspectWord + ", " + suspectCode + "," + suspectAltCode + " and " +
matchingWord + ", " + matchingCode + ", " + matchingAltCode);
     matchedWords[numberSimilarSoundingWords] = new Integer(matchingWordNumber);
     numberSimilarSoundingWords++;
     lastMatchingWord = matchingWord;
     break; //so that there is no possibility of the suspect word matching more than once
    }
    matchingWordNumber++;
   }
  }
  //if all words sound alike
  if (numberOfWordsInSuspectString == numberOfWordsInMatchingString
   && numberOfWordsInSuspectString == numberSimilarSoundingWords) {
    allWordsSoundAlike = true;
  }
  else {
    allWordsSoundAlike = false;
  }
  _log.info("returning from allWordsSoundAlike : " + allWordsSoundAlike);
```

```
 return allWordsSoundAlike;
}
/** Scores the words in the strings. This is a very important method, tamper with it at your peril.
 * @param String originalName The first of the two strings to check
 * @param String suspectString The first string that is checked against
 * @param String name The second of the two strings to check
 * @param String type This is used to try and better match strings where the surname can be distinguished.
 * Currently this is only possible for 'Individuals' on the 'SDN' list.
 * @param String listType The list that the hit was on, e.g. 'SDN'.
 * @return double score for the match between the two words. Always a value between zero and one hundred.
 * @since 2.1
 */
  public double matchScores(String unadulteratedMatchingName, String inString, String matchingName, String type,
String listType, String hitWord, float hitWordScore, int incomingWordNum) {
 _log.info("matchscores : original name " + unadulteratedMatchingName + " inString :" + inString + " matchingName :"
+ matchingName + " type :" + type + " listType :" + listType +  " hitWord " + hitWord + " hitWordScore " + hitWordScore
+  " incomingWordNum " + incomingWordNum + "<");
 int numberOfWordsInMatchingString = calcNumberOfWordsInString(matchingName);
 int numberOfWordsLeftInMatchingString = numberOfWordsInMatchingString;
 int numberOfWordsInIncomingString = calcNumberOfWordsInString(inString);
 int numberOfWordsLeftInIncomingString = numberOfWordsInIncomingString;
 double matchScore = 0;
 boolean surnameFound = false;
 boolean surnameOriginallyHit = false;
 float surnameModifierFactor = (float) 0.0;
 String[] incomingWords = new String [numberOfWordsInIncomingString];
 String[] matchingWords = new String [numberOfWordsInMatchingString];
 Vector[] matchingWordsCodes = new Vector[numberOfWordsInMatchingString];
 Integer[] incomingWordsHit = new Integer [numberOfWordsInIncomingString];
 Integer[] matchingWordsHit = new Integer [numberOfWordsInMatchingString];
 Integer[] incomingWordsHitMatchingWord = new Integer [numberOfWordsInIncomingString];
 float allGoodHitsWordScores = hitWordScore;;
 int surnameBarrels = 0;
 /*
 ** intelligent surname matching is available only for certain types of match. Currently this is limited
 ** to Individuals in the SDN lists. Intelligent name matching can be turned on or off in the Filter.properties
 ** file.
 */
 if (type.equals(Constants.INDIVIDUAL_TYPE_CODE) && listType.equals(Constants.SDN_LIST_TYPE_CODE) &&
intelligentSurnameMatching.equals("on")) {
  int surnameOffset = unadulteratedMatchingName.indexOf(",");
  if (surnameOffset > 0) {
   String surname =  unadulteratedMatchingName.substring(0, surnameOffset);
   if (surname != null && !surname.equals("")) {
    StringTokenizer srn = new StringTokenizer(surname);
    surnameBarrels = srn.countTokens();
    surnameFound = true;
    _log.info("multi barelled surname, " + surname + ", found. It has " + surnameBarrels + " barrels");
    /*
    ** some checking needs to be done to see that the surname is still the same after
   ** short & stopwords have been removed. If anything has been removed then the
   ** surname matching is not carried out.
   */
   //Check if the surname has already been scored as the original word hit.
   if (surname.indexOf(hitWord) >= 0) {
    surnameOriginallyHit = true;
    hitWordScore = hitWordScore * (float)SURNAMEWEIGHT;
    _log.info("surname already scored as original hit, modified to " + hitWordScore);
   }
   //hit must have been on a forename word
   else {
    hitWordScore = hitWordScore * (float)PRENOMWEIGHT;
    _log.info("forename already scored as original hit, modified to " + hitWordScore);
   }
   allGoodHitsWordScores = hitWordScore;
   surnameModifierFactor = surnameBarrels*(float)0.2;
   if (numberOfWordsInMatchingString < numberOfWordsInIncomingString) {
    _log.info("surname modifier factor, shorter matching string " + surnameModifierFactor);
    surnameModifierFactor = surnameModifierFactor - (numberOfWordsInMatchingString + 1 -
surnameBarrels)*(float)0.2;
   }
   else {
    _log.info("surname modifier factor longer matching string " + surnameModifierFactor);
    surnameModifierFactor = surnameModifierFactor - (numberOfWordsInIncomingString + 1 -
surnameBarrels)*(float)0.2;
   }
   _log.info("surname modifier factor " + surnameModifierFactor);
  }
 }
}
 boolean goodHit = false;
 int nextWordinStringStartPosition = 0;
 int wordCount = 0;
 /*
 ** make an array of the incoming words
 */
 StringTokenizer stIncomingString = new StringTokenizer(inString);
 int incomingWordCount = 0;
 while (stIncomingString.hasMoreTokens()) {
  incomingWords[incomingWordCount] = stIncomingString.nextToken();
  incomingWordsHit[incomingWordCount] = new Integer(0);
  incomingWordCount++;
 }
 /*
 ** make an array of the matching words
 */
 StringTokenizer stMatchingString = new StringTokenizer(matchingName);
 int matchingWordCount = 0;
 int hitWordNum = 0;
 while (stMatchingString.hasMoreTokens()) {
  matchingWords[matchingWordCount] = stMatchingString.nextToken();
  if (hitWord.equals(matchingWords[matchingWordCount])) {
   hitWordNum = matchingWordCount;
  }
  //optimization - if only one word in both strings then don't bother generating the code
  if (numberOfWordsInMatchingString > 1 || numberOfWordsInIncomingString > 1) {
   matchingWordsCodes[matchingWordCount] =
com.sybase.patriotact.utils.Match.altCodeV(matchingWords[matchingWordCount]);
  }
  matchingWordsHit[matchingWordCount] = new Integer(0);
  matchingWordCount++;
 }
 //go through the incoming words array trying to find matches in the matching words array
 boolean incomingAndMatchingWordMatch = false;
 for (int i = 0; i < numberOfWordsInIncomingString; i++ ) {
  incomingAndMatchingWordMatch = false;
  //optimization, if only one word left in both strings then ignore codes & do the match now
  if (numberOfWordsInMatchingString == 1 && numberOfWordsInIncomingString == 1) {
   matchScore = ((double)com.sybase.patriotact.utils.Match.score(incomingWords[0], matchingWords[0])/10);
   if (surnameFound == true) {
    if (((i < surnameBarrels) && surnameOriginallyHit == false) ||
     ((i < surnameBarrels - 1) && surnameOriginallyHit == true)) {
     _log.info ("surname matching " + incomingWords[0] + " to " + matchingWords[0] + " new score " +
matchScore*SURNAMEWEIGHT + " old score " + matchScore);
     //Weigh surnames more
     matchScore = matchScore*SURNAMEWEIGHT;
    }
    else {
     _log.info ("first name matching (in surname) " + incomingWords[0] + " to " + matchingWords[0]  + " new score " +
matchScore*PRENOMWEIGHT + " old score " + matchScore);
     //Weigh firstnames less
     matchScore = matchScore*PRENOMWEIGHT;
    }
   }
   allGoodHitsWordScores = allGoodHitsWordScores + (float) matchScore;
   //mark these two words as hit
   incomingWordsHit[0] = new Integer(1);
   matchingWordsHit[0] = new Integer(1);
   incomingWordsHitMatchingWord[0] = new Integer(0);//always the first word hit the first word
   numberOfWordsLeftInIncomingString = 0;
   numberOfWordsLeftInMatchingString = 0;
  }
  else {
   //get the codes for this word
   Vector incomingWordCodes = com.sybase.patriotact.utils.Match.altCodeV(incomingWords[i]);
   String incomingWordCode = (String)incomingWordCodes.elementAt(0);
   String incomingWordAltCode = (String)incomingWordCodes.elementAt(1);
   boolean wordHit = false;
   for (int j = 0; j < numberOfWordsInMatchingString; j++) {
    //Check if matching word already hit
    if (matchingWordsHit[j].intValue() == 1) {
     continue;
    }
    String matchWord = matchingWords[j];
    //Optimization - If the word is exactly the same, then don't bother with codes or scores.
    if (incomingWords[i].equals(matchingWords[j])) {
     matchScore = (double)100.00;
     _log.info("matching exactly two words, " + incomingWords[i] + " and " + matchWord + " found.");
     //allGoodHitsWordScores = allGoodHitsWordScores + (float) matchScore;
     //mark these two words as hit
     incomingWordsHit[i] = new Integer(1);
     matchingWordsHit[j] = new Integer(1);
     incomingWordsHitMatchingWord[i] = new Integer(j);
     numberOfWordsLeftInIncomingString--;
     numberOfWordsLeftInMatchingString--;
     wordHit = true;
    }
    else {
     String matchingWordCode = (String)((matchingWordsCodes[j]).elementAt(0));
     String matchingWordAltCode = (String)((matchingWordsCodes[j]).elementAt(1));
     if ( incomingWordCode.equals(matchingWordCode) ||
      incomingWordCode.equals(matchingWordAltCode) ||
      incomingWordAltCode.equals(matchingWordCode) ||
      incomingWordAltCode.equals(matchingWordAltCode)) {
      //the two words have a code in common so score them
      matchScore = ((double)com.sybase.patriotact.utils.Match.score(incomingWords[i], matchWord)/10);
      _log.info("matching codes for two words, " + incomingWords[i] + " and " + matchWord + " found.");
      //allGoodHitsWordScores = allGoodHitsWordScores + (float) matchScore;
      //mark these two words as hit
      incomingWordsHit[i] = new Integer(1);
      matchingWordsHit[j] = new Integer(1);
      incomingWordsHitMatchingWord[i] = new Integer(j);
      numberOfWordsLeftInIncomingString--;
      numberOfWordsLeftInMatchingString--;
      //a hit means don't try any more matches for this incoming word
      incomingAndMatchingWordMatch = true;
      wordHit = true;
     }
    }
    if (wordHit == true) {
     if (surnameFound == true) {
      if (((j < surnameBarrels) && surnameOriginallyHit == false) ||
       ((j < surnameBarrels - 1) && surnameOriginallyHit == true)) {
       _log.info ("surname matching " + incomingWords[i] + " to " + matchingWords[j] + " new score " +
matchScore*SURNAMEWEIGHT + " old score " + matchScore);
       //Weigh surnames more
       matchScore = matchScore*SURNAMEWEIGHT;
      }
      else {
       _log.info ("first name matching (in surname) " + incomingWords[i] + " to " + matchingWords[j]  + " new score " +
matchScore*PRENOMWEIGHT + " old score " + matchScore);
```

```
       //Weigh firstnames less
       matchScore = matchScore*PRENOMWEIGHT;
      }
     }
     allGoodHitsWordScores = allGoodHitsWordScores + (float) matchScore;
     break;
    }
   }
  }
 }
 _log.info("numberOfWordsLeftInIncomingString " + numberOfWordsLeftInIncomingString + "
numberOfWordsLeftInMatchingString " + numberOfWordsLeftInMatchingString);
 /*
 ** Go back over the incoming words & score all the non-matched words to all the remaining words
 ** in the matching string to form a cartesian product of matches which will be later sifted to find
 ** the best scores.
 */
 int numberRemainingMatchingWordsCartProd =
numberOfWordsLeftInIncomingString*numberOfWordsLeftInMatchingString;
 Double[] wordScores = new Double [numberRemainingMatchingWordsCartProd];
 Integer[] incomingWordNumbers = new Integer[numberRemainingMatchingWordsCartProd];
 Integer[] matchWordNumbers = new Integer[numberRemainingMatchingWordsCartProd];
 for (int i = 0; i < incomingWordCount; i++ ) {
  //Check if matching word already hit
  if (incomingWordsHit[i].intValue() == 1) {
   continue;
  }
  for (int j = 0; j < matchingWordCount; j++) {
   //Check if matching word already hit
   if (matchingWordsHit[j].intValue() == 1) {
    continue;
   }
   matchScore = ((double)com.sybase.patriotact.utils.Match.score(incomingWords[i], matchingWords[j])/10);
   _log.info("matching " + incomingWords[i] + " to " + matchingWords[j] + " matched " + matchScore + "% " +
wordCount);
   wordScores[wordCount] = new Double(matchScore);
   incomingWordNumbers[wordCount] = new Integer(j);
   matchWordNumbers[wordCount] = new Integer(i);
   wordCount++;
  }
 }
 //Decide how many words are important in the match
 float numberOfWordsToMatch = 0;
 //number of words to consider in the match is the lesser of the number of words in either of the two strings
 if (numberOfWordsInIncomingString < numberOfWordsInMatchingString) {
  numberOfWordsToMatch = (float)numberOfWordsInIncomingString;
 }
 else {
  numberOfWordsToMatch = (float)numberOfWordsInMatchingString;
 }
 //If the match is from the cleared list fuzzy match then there is no extra hitword to consider
 if (!hitWord.equals("")) {
  numberOfWordsToMatch = numberOfWordsToMatch + surnameModifierFactor + 1;
 }
 _log.info("number of words to consider in matching " + numberOfWordsToMatch);
 //Declare arrays to hold the best matches
 Integer[] bestIncomingWords = new Integer[numberOfWordsLeftInIncomingString];
 Integer[] bestMatchWords = new Integer[numberOfWordsLeftInMatchingString];
 Double[] bestScores = new Double [numberOfWordsLeftInIncomingString];
 boolean ignoreScore = false;
 int validWordHits = 0;
 for (int j = 0; j < numberOfWordsLeftInIncomingString; j++) {
  _log.info("Scanning results iteration : " + j);
  double bestScore = 0.0;
  int bestword = 0;
  //For each row in the score matrix
  for (int i = 0 ; i < numberRemainingMatchingWordsCartProd; i++) {
   _log.info ("score for word " +  matchWordNumbers[i]  + " " + wordScores[i]);
   //Ignore the best existing match
   for (int k = 0; k < j; k++) {
    if (matchWordNumbers[i].intValue() == bestMatchWords[k].intValue() ||
     incomingWordNumbers[i].intValue() == bestIncomingWords[k].intValue()) {
     _log.debug ("ignoring word " + i + ", it's already got a best score.");
     ignoreScore = true;
     break;
    }
    else {
     ignoreScore = false;
    }
   }
   if (wordScores[i].doubleValue() >= bestScore && ignoreScore == false) {
    bestIncomingWords[j] = incomingWordNumbers[i];
    bestMatchWords[j] = matchWordNumbers[i];
    bestScores[j] = new Double (wordScores[i].doubleValue());
    incomingWordsHitMatchingWord[j] = new Integer(j);
    //RESOLVE - this is probably where more incoming words are marked as hitting match words for consecutive word
hits
    bestScore = wordScores[i].doubleValue();
    _log.info ("Best matching word for suspect word " + incomingWordNumbers[i] + " is matched word " +
matchWordNumbers[i] + " at " + " score " + wordScores[i].doubleValue());
   }
  }
 }
 //Retrieve just the correct number of best scores
 int numberOfWordsInResultSet = 0;
 if (numberOfWordsLeftInIncomingString < numberOfWordsLeftInMatchingString) {
  numberOfWordsInResultSet = numberOfWordsLeftInIncomingString;
 }
 else {
  numberOfWordsInResultSet = numberOfWordsLeftInMatchingString;
 }
 _log.info ("Number of words in result set " + numberOfWordsInResultSet);
 //Get the final score from the array of best scores. Only retrieve the appropriate number of best scores
 matchScore = 0.0;
 for (int i = 0 ; i < numberOfWordsInResultSet; i++) {
  _log.info("totaling matchScore " + bestScores[i].doubleValue());
  matchScore = bestScores[i].doubleValue() + matchScore;
 }
 //Add the word score for the word that originally hit plus matches where the codes were the same, to the total score for
all the other words
 matchScore = matchScore + allGoodHitsWordScores ;
 _log.info ("finally in matchScore : matchWordScore " + matchScore + " hitWordScore " + hitWordScore);
 matchScore = matchScore / numberOfWordsToMatch ;
 /*
 ** First cut at implementing word proximity matching.
 **
 */
 if (!type.equals(Constants.INDIVIDUAL_TYPE_CODE) && numberOfWordsInIncomingString > 1) {
  boolean twoConsecutiveWordsHit = false;
  boolean oneIncomingWordHit = false;
  boolean twoIncomingWordHit = false;
  //First prove if two consecutive incoming words hit
  for (int i = 0; i < numberOfWordsInIncomingString; i++) {
   //System.out.println(i + " " + incomingWords[i] + " matched to " + incomingWordsHitMatchingWord[i] + " " +
incomingWordsHit[i]);
   if (incomingWordsHitMatchingWord[i] != null && oneIncomingWordHit == true) {
    twoIncomingWordHit = true;
    break;
   }
   if (incomingWordsHitMatchingWord[i] != null && (incomingWordNum == i)) {
    twoIncomingWordHit = true;
    break;
   }
   if (incomingWordsHitMatchingWord[i] != null) {
    oneIncomingWordHit = true;
   }
   else {
    oneIncomingWordHit = false;
   }
  }
  if (twoIncomingWordHit == false) {
   matchScore = matchScore * lowMatchThresholdValue;
   _log.info("because there was not at least two consecutive words hit the score has been diminished");
  }
 }
 return matchScore;
   }
 /**
 * @param inString String
 * @param noSigficantwords integer
 * @return String truncated to only the 'noSigficantwords'. This is typically used to truncate long names before
 * searching the cleared list for a fuzzy cleared list search.
 * @since 2.1.2
 */
 private String getOnlyFirstFewWords(String inString, int noSigficantWords) {
  StringBuffer outBuf = new StringBuffer();
  StringTokenizer st = new StringTokenizer(inString);
  int wordCount = 0;
  while (st.hasMoreTokens()) {
   outBuf.append(st.nextToken() + " ");
   wordCount++;
   if (wordCount >= noSigficantWords) {
    break;
   }
  }
  _log.info("return from getOnlyFirstFewWords : " + outBuf.toString() + "<");
  return outBuf.toString();
 }
 /** Adds SQL wildcard characters to a string.
 * E.g. the string 'test string' will be returned as '%test%string%'. This is typically used for fuzzy cleared list
 * matching. (Not true fuzzy matching but the wildcards will compensate for small spelling differences.)
 * @param inString String
 * @return String wildcarded
 * @since 2.1
 */
 private String addWildCards(String inString) {
  StringBuffer outBuf = new StringBuffer();
  StringTokenizer st = new StringTokenizer(inString);
  outBuf.append(Constants.WILDCARD);
  while (st.hasMoreTokens()) {
   outBuf.append(st.nextToken() + Constants.WILDCARD);
  }
  return outBuf.toString();
 }
 /** Return a String with the order of the words reversed. Useful in building the SQL string for
 * fuzzy cleared list searching when the search string is to be reversed, e.g. 'Homer Simpson' becomes 'Simpson
Homer'.
 * @param inString string to be reversed
 * @param offset The word number in the string from which to reverse the words.
 * @return The reversed string
 * @since 2.1
 */
 private String reverseWordsInString(String inString, int offset) {
  StringBuffer outBuf = new StringBuffer();
  StringTokenizer st = new StringTokenizer(inString);
  ArrayList words = new ArrayList ();
  while (st.hasMoreTokens()) {
   words.add(st.nextToken());
  }
  int i = words.size();
  i = i - 1;
  if (i == 0) {
   //complete reversal of words
   while (i >= 0) {
    outBuf.append(words.get(i) + " ");
    i--;
   }
  }
  else {
   //toggle & reversal of words
   while (i >= offset) {
    outBuf.append(words.get(i) + " ");
    i--;
   }
   int j = 0;
   while (j < offset) {
    outBuf.append(words.get(j) + " ");
    j++;
   }
  }
  return outBuf.toString().trim();
 }
 /*
 * main entry point for in-memory matching.
 * @param Object record to be searched.  This is only needed for distinguishing the type of the object.
 * @param String name to be searched
 * @param java.sql.Connection conn Connection to the database
 * @return Arraylist of matches
 * @since 2.1.1
 */
 public ArrayList generateCodes(Object record, String matchingField, String name, String clearedListSqlString,
java.sql.Connection conn) throws PatriotSearchException {
  _log.info("generateCode : record " + record + " matchingField " + matchingField + " name " + name + "
clearedListSqlString " + clearedListSqlString + " Connection " + conn);
  ArrayList suspectHitResults = new ArrayList(); //Arrray for adding return values
  try {
   String workingName = removeScoreStopWords(removeNonAlphabeticCharacters(name));
   //If there's nothing left then reuse the original string
   if (workingName == null || workingName.equals("")) {
    workingName = removeNonAlphabeticCharacters(name);
   }
   StringTokenizer wn = new StringTokenizer(workingName);
   int numberOfWordsInWorkingString = wn.countTokens();// = 0; RESOLVE - don't need this if the number of words to
divide by is always two.
   ArrayList matchingEntIds = new ArrayList();
   int incomingWordNum = 0;
   int wordCount = 0;
   int validWordCount = 0;
   while (wn.hasMoreTokens()) {
    wordCount++;
    //Get the next word in the string
    String word = wn.nextToken().trim();
    /*
    ** Short words are removed from matching, unless the string consists of only short words
    ** in which ignore all but the last short word.
    */
    //Remove short words
    if (word.length() <= 2 ) {
     //Don't try matching unless it's the last word
     if (wordCount < numberOfWordsInWorkingString) {
      _log.info("discarding a short word");
      continue;
     }
     //Don't match if it's the last word and at least one previous word has been searched
     if ((wordCount == numberOfWordsInWorkingString) && (validWordCount > 0)) {
      _log.info("discarding the last short word");
      continue;
     }
    }
    validWordCount++;
    Vector codes = com.sybase.patriotact.utils.Match.altCodeV(word);
    String code = (String) codes.elementAt(0);
    String altCode = (String) codes.elementAt(1);
    _log.info("testing code " + code + "<");
    /*
    ** This list holds all the entid's already searched for the current word.
    ** If more than one word matches then the name will not be searched again.
    */
    //ArrayList alreadyTestedEntIds = new ArrayList();
    //ArrayList alreadyTestedAltNums = new ArrayList();
    //ArrayList alreadyTestedSuspectWords = new ArrayList();
    suspectHitResults.addAll(matchString (record, matchingField, code, word, workingName, name,
numberOfWordsInWorkingString, matchingEntIds, clearedListSqlString, incomingWordNum, conn));
    if (!code.equals(altCode)) {
     _log.info("testing altCode " + altCode + "<");
     suspectHitResults.addAll(matchString (record, matchingField, altCode, word, workingName, name,
numberOfWordsInWorkingString, matchingEntIds, clearedListSqlString, incomingWordNum, conn));
    }
    incomingWordNum++;
   }
  }  catch (Exception e) {
      e.printStackTrace();
   throw new PatriotSearchException();
  }
  return suspectHitResults;
 }
 /*
 ** The main matching code is contained here.
 */
 public ArrayList matchString (Object record, String matchingField, String code, String word, String workingName,
String name, int numberOfWordsInWorkingString, ArrayList matchingEntIds, String clearedListSqlString, int
```

```
incomingWordNum, java.sql.Connection conn) throws PatriotSearchException {
  try {
   boolean hasCode = false;
   ArrayList suspectHitResults = new ArrayList(); //Array for adding return values
   //Check if the code matches anything in the list of suspects
   if (Class.forName("com.sybase.patriotact.filter.DbRecord").isAssignableFrom(record.getClass()) && !(record
instanceof OnlineCustomerObject) && !(record instanceof OnlineEmployeeObject)) {
    hasCode = this.searchHash.quickMatches.containsKey(code);
   }
   else {
    //Make sure the service component has loaded the data, if not loaded then load it
    if (LoaderImpl.quickMatches == null) {
     loadingSemaphore = true;
     LoaderImpl.reload();
     loadingSemaphore = false;
    }
    while (loadingSemaphore == true) {
     java.lang.Thread.sleep(1000);
     System.out.println("Sleeping while the in-memory suspects array is refreshed...");
    }
    hasCode = LoaderImpl.quickMatches.containsKey(code);
   }
   if (hasCode == true) {
    _log.info("prematched to at least one suspect...");
    //Get all the words that have this code. NightlyFilter uses it's own hashTable.
    QuickMatchObject o = null;
    if (Class.forName("com.sybase.patriotact.filter.DbRecord").isAssignableFrom(record.getClass()) && !(record
instanceof OnlineCustomerObject) && !(record instanceof OnlineEmployeeObject)) {
     o = (QuickMatchObject)this.searchHash.quickMatches.get(code);
    }
    else {
     o = (QuickMatchObject)LoaderImpl.quickMatches.get(code);
    }
    SuspectHitResult suspectHit = null;
    boolean thisRowMatchedThisSuspectAlready = false;
    boolean isAlreadyMatched = false;
    List a =  null;
    a = o.getQuickMatchWords();
    //Get the entIds
    List id = null;
    id = o.getQuickMatchEntIds();
    //Get the list_types
    List listTypes = null;
    listTypes = o.getQuickMatchListTypes();
    //Get the types
    List types = null;
    types = o.getQuickMatchTypes();
    String lastMatchingWord = "";
    String lastWord = "";
    //int lastEntId = 0;
    //List narrows, only for MoneyTransactions.
    ArrayList listNarrows = null;
    if (record instanceof MoneyTransactionObject) {
     //Get the list.
     listNarrows = ((MoneyTransactionObject)record).getScanList();
    }
    //For every word in the list of suspects that has this code
    for (int i = 0; i < a.size(); i++) {
     //List narrows, only for MoneyTransactions
     if (record instanceof MoneyTransactionObject) {
      //if there is a subset of lists to match against
      if (listNarrows.isEmpty() == false) {
       boolean narrowListHit = false;
       for (int j = 0; j < listNarrows.size(); j++) {
        //see if the match was against a subset list
        if (((String)listTypes.get(i)).trim().equals(listNarrows.get(j))) {
         narrowListHit = true;
         break;
        }
       }
       if (narrowListHit == false) {
        //there was no hits against the subset
        _log.info("The potential hit was not in the sublist, it will be ignored");
        continue;
       }
      }
     }
     String matchingWord = (String)a.get(i);
     Integer entId = (Integer)id.get(i);
     //if the last match was on the same word pair & it failed then don't rescore
     if (matchingWord.equals(lastMatchingWord) && word.equals(lastWord)) {
      continue;
     }
     /*
     ** If this word has already been searched for this ent_id & this altnum then don't search again.
     */
     /*boolean thisRecordAlreadySearched = false;
     for (int j = 0; j < alreadyTestedEntIds.size(); j++) {
      _log.info("compare " + entId + " and " + alreadyTestedEntIds.get(j) + " for previous searching");
      //System.out.println("entid " + entId + " alreadyTestedEntIds " + alreadyTestedEntIds.get(j) + "
o.getQuickMatchAltNum " + o.getQuickMatchAltNum(i) + " alreadyTestedAltNums " + alreadyTestedAltNums.get(j) + "
o.getQuickMatchSuspectWords(i) " + o.getQuickMatchSuspectWords(i) + " alreadyTestedSuspectWords " +
alreadyTestedSuspectWords.get(j));
      //System.out.println("entid " + entId + " alreadyTestedEntIds " + alreadyTestedEntIds.get(j) + "
o.getQuickMatchSuspectWords(i) " + o.getQuickMatchSuspectWords(i) + " alreadyTestedSuspectWords " +
alreadyTestedSuspectWords.get(j));
      if (entId.compareTo((Integer)alreadyTestedEntIds.get(j)) == 0 &&
       //o.getQuickMatchAltNum(i).compareTo((Integer)alreadyTestedAltNums.get(j)) == 0 &&
       o.getQuickMatchSuspectWords(i).equals((String)alreadyTestedSuspectWords.get(j))) {
        thisRecordAlreadySearched = true;
     }
    }
    if (thisRecordAlreadySearched == true) {
     _log.info("already had a search on entid " + entId  + " and altNum " + o.getQuickMatchAltNum(i) + " and word " +
o.getQuickMatchSuspectWords(i));
     continue;
    }*///RESOLVE - BE CAREFUL HERE THAT THIS DOES NOT BREAK THE NAME TRANSATION
     //alreadyTestedEntIds.add(entId);
     //alreadyTestedAltNums.add(o.getQuickMatchAltNum(i));
     //alreadyTestedSuspectWords.add(o.getQuickMatchSuspectWords(i));
    _log.info("investigating " + word + " with " + matchingWord + " for entId " + entId + " and altNum " +
o.getQuickMatchAltNum(i));
    //if the last match was on the same word pair & it failed then don't rescore
    //if (matchingWord.equals(lastMatchingWord) && word.equals(lastWord)) {
     //continue;
    //}
    /*
    ** If there's already a case for this suspect then don't go any further.
    ** This could happen if there is more than one pre-matching word in the names.
    */
    thisRowMatchedThisSuspectAlready = false;
    for (int j = 0; j < matchingEntIds.size(); j++) {
     _log.info("compare  " + entId + " and " + matchingEntIds.get(j) + " for previous matching");
     if (entId.compareTo((Integer)matchingEntIds.get(j)) == 0) {
      thisRowMatchedThisSuspectAlready = true;
     }
    }
    if (thisRowMatchedThisSuspectAlready == true) {
     _log.info("already have a match for entid " + entId);
     continue;
    }
    //Optimization - If the word is exactly the same, then don't bother to score.
    double matchingWordScore = 0.0;
    if (matchingWord.equals(word)) {
     matchingWordScore = (double)1000.00;
    }
    else {
     matchingWordScore = com.sybase.patriotact.utils.Match.score(matchingWord, word);
    }
    _log.info("matchingWordScore " + matchingWordScore + " matchThresholdFromScore " +
matchThresholdFromScore);
    //If the score for this one word match is good enough to warrant more investigation of the names
    if (matchingWordScore > goodSoundingMatchOverrideThresholdFromScore) {
     /*
     ** At this point a more thorough examination of the two strings will take place.
     ** So mark the two names as searched.
     */
     //alreadyTestedEntIds.add(entId);
     //alreadyTestedAltNums.add(o.getQuickMatchAltNum(i));
     //alreadyTestedSuspectWords.add(o.getQuickMatchSuspectWords(i));
    //divide the score now
    matchingWordScore = matchingWordScore/10;
    //Remove this word from the string since it's already scored, before passing it to the matchscores method
    int wordStartsAt = workingName.indexOf(word);
    String shortWorkingName = workingName.substring(0, wordStartsAt) + " " +
workingName.substring(wordStartsAt+word.length());
    //Get the name for this matching word
    String matchingName = null;
    String unadulteratedMatchingName = o.getQuickMatchName(i);
    matchingName = removeNonAlphabeticCharacters(unadulteratedMatchingName);
    int matchingWordStartsAt = matchingName.indexOf(matchingWord);
    String shortMatchingName = "";
    //There's a small chance that the word will not be in the name, if the name was longer in PatMasterList than
PatQuickMatch allows
    if (matchingWordStartsAt < 0) {
     shortMatchingName = matchingName;
    }
    else {
     shortMatchingName = matchingName.substring(0, matchingWordStartsAt) + " " +
matchingName.substring(matchingWordStartsAt+matchingWord.length());
    }
    //get rid of short words on the matching name
    shortMatchingName = removeShortWords(shortMatchingName);
    //get rid of stop words on the matching name
    shortMatchingName = removeScoreStopWords(shortMatchingName);
    double totalScore = 0;
    //If there's only one word left then there's no point in doing any more matching. Also if it's a concatenated word name
hit.
    if (shortWorkingName.trim().length() == 0 || ((shortMatchingName.trim().length() == 0) ||
(o.getQuickMatchOriginalNameInd(i).intValue() == 2))) {
     //one word in both strings, e.g. 'EGYPT' to 'EGYPT'. Ignore lowscorethreshold. Also if it's a concatenated word hit.
     if (shortWorkingName.trim().length() == 0 && shortMatchingName.trim().length() == 0) {
      totalScore = matchingWordScore;
     }
     //or if it's a concatentated word name e.g. 'dinoarmani'
     else if (o.getQuickMatchOriginalNameInd(i).intValue() == 2 && shortWorkingName.trim().length() == 0) {
      totalScore = matchingWordScore;
     }
     /*
     ** Score differently hits to the FATF or keyWord list if the hit name only has one word. This is to allow hits on phrases
like
     ** 'send the money to egypt' without having keyword matching turned on.
     */
     else if ((((String)listTypes.get(i)).trim().equals(Constants.FATF_LIST_TYPE_CODE) ||
      ((String)listTypes.get(i)).trim().equals(customKeywordList)) && shortMatchingName.trim().length() == 0) {
      _log.info("A hit was detected to a FATF list entry that only has one word in the name.");
      totalScore = matchingWordScore;
     }
     else {
      //totalScore = matchingWordScore * lowMatchThresholdValue;
```

```
        String hitWordIndicator = hitOnSurname(o.getQuickMatchName(i), matchingWord, ((String)types.get(i)).trim(),
((String)listTypes.get(i)).trim(), intelligentSurnameMatching);
        if(hitWordIndicator.equals("forenameHit")) {
         //one word & it hit a forename then degrade the score twice
         totalScore = matchingWordScore * lowMatchThresholdValue * lowMatchThresholdValue;
        }
        else if(hitWordIndicator.equals("surnameHit")) {
         totalScore = matchingWordScore;
        }
        else { //undefined hit
         totalScore = matchingWordScore * lowMatchThresholdValue;
        }
       }
      }
      else {
       //get rid of short words on the matching name
       //shortMatchingName = removeShortWords(shortMatchingName);
       _log.info("names " + shortWorkingName + " with " + shortMatchingName);
       /*
       ** Exception - if both strings have more than (5) words then at least two must prematch.
       ** Since one has already prematched to get this far we're looking for just one more match.
       */
       if (calcNumberOfWordsInString(shortWorkingName) >= thresholdForMultipleSoundAlikes) {
        if (calcNumberOfWordsInString(shortMatchingName) >= thresholdForMultipleSoundAlikes) {
         _log.info("Checking two long strings : " + shortWorkingName + " : and : " + shortMatchingName);
         int numSoundAlikeWords = 0;
         StringTokenizer wn = new StringTokenizer(shortWorkingName);
         while (wn.hasMoreTokens()) {
          StringTokenizer mn = new StringTokenizer(shortMatchingName);
          //Get the next word in the in string
          String inWord = wn.nextToken();
          //Get the codes for the instring
          Vector inCodes = com.sybase.patriotact.utils.Match.altCodeV(inWord);
          String inCode = (String) inCodes.elementAt(0);
          String altInCode = (String) inCodes.elementAt(1);
          while (mn.hasMoreTokens()) {
           //Get the next word in the matching string
           String longStringMatchingWord = mn.nextToken();
           Vector matchingCodes = com.sybase.patriotact.utils.Match.altCodeV(longStringMatchingWord);
           String matchingCode = (String) matchingCodes.elementAt(0);
           String altMatchingCode = (String) matchingCodes.elementAt(1);
           if (inCode.equals(matchingCode) || inCode.equals(altMatchingCode)
            || altInCode.equals(matchingCode) || altInCode.equals(altMatchingCode)) {
            numSoundAlikeWords++;
            //if (numSoundAlikeWords >= 2) {
             break;//Job done, at least two are a good match
            //}
           }
          }
         }
        //If no more words matched then ignore this pre-match
        if (numSoundAlikeWords < 1) {
         _log.info("Only one pre-matching word was found for two long strings, so the match is being ignored");
         continue;
        }
       }
      }
       //Get scoring for all the other words
      double otherWordScores = matchScores(o.getQuickMatchName(i), shortWorkingName, shortMatchingName,
((String)types.get(i)).trim(), ((String)listTypes.get(i)).trim(), matchingWord, (float)matchingWordScore,
incomingWordNum);
      /*
      ** It's always divide by two because the two components of score are
      ** already adjusted for the number of words, i.e. matchingWordScore is always for 1 word
      */
      totalScore = otherWordScores; //(otherWordScores + matchingWordScore)/2;
      _log.info("resultant totalscores " + totalScore);
     }
     /*
     ** score differently matches from one word country fields to non-country types of more than one word.
     ** This could be just one long 'if' statement.
     ** The reason for this is, say a field with value 'FRANCE' is being searched & there is a suspect 'JAMES FRANCE'.
     ** That would hit otherwise.
     */
     if (matchingField.indexOf(Constants.COUNTRY_SEARCH) > -1) {
      if ( numberOfWordsInWorkingString == 1  && !((String)types.get(i)).trim().equals(Constants.COUNTRY_TYPE_CODE)
&& calcNumberOfWordsInString(shortMatchingName) > 0) {
       totalScore = totalScore * lowMatchThresholdValue;
       _log.info("modifying the score a one-word country field match to a non-country suspect with more than one word. " +
totalScore);
      }
     }
     //If it's a good score or close to a good score
     if (totalScore > goodSoundingMatchOverrideThreshold) {
      boolean allWordsSoundAlike = false;
      //If it's close try a good-sounding-match override
      if ((totalScore < scoreThreshold)
       && ( (totalScore > goodSoundingMatchOverrideThreshold) && goodSoundingMatchOverride.equals("on") )) {
       allWordsSoundAlike = allWordsSoundAlike(workingName, matchingName);
       if (allWordsSoundAlike == false) {
        continue;//it's not a match so go to the next match
       }
      }
      //Either it's a match above the threshold or it just missed the threshold but the words all sound alike
      if (totalScore > scoreThreshold || allWordsSoundAlike == true) {
       int unadulteratedMatchingNameInd = o.getQuickMatchOriginalNameInd(i).intValue();
       int altNum = o.getQuickMatchAltNum(i).intValue();
       //Don't go to the cleared list if it's a a ClearList table search from the NightlyFilter
       if (!(record instanceof ClearedListObject)) {
        suspectHit = inClearedList(unadulteratedMatchingName, name, clearedListSqlString, entId.longValue(), altNum,
unadulteratedMatchingNameInd, conn);
```

```
        }
        else {
         suspectHit = furnishHitDetails (unadulteratedMatchingName, name, entId.longValue(), altNum,
unadulteratedMatchingNameInd, ((String)listTypes.get(i)).trim(), conn);
        }
        if (suspectHit != null) {
         //Report if the match was actually against an alias name
         if (altNum > 0) {
          _log.warn("The match was on alias " + matchingName + " of the suspect ");
         }
         /*
         ** It's confusing for users to have hits reported that are below the threshold, (for example with a 'good-sounding-
match') so
         ** to avoid this simply elevate the score to the thresohld if required.
         */
         if (totalScore < scoreThreshold) {
          _log.info("The score was below the score threshold but it will be elevated to avoid confusion in the dispalyed results.");
          totalScore = scoreThreshold;
         }
         //Stop the score getting displayed to upteen decimal places
         String totalScoreString = String.valueOf(totalScore);
         int totalScoreStringLength = totalScoreString.length();
         if (totalScoreString.length() > 5) {
          totalScoreStringLength = 5;
         }
         _log.warn("Raise " + workingName + " matching " + matchingName + " (" + entId + ") scores " +
String.valueOf(totalScore).substring(0,totalScoreStringLength));
         suspectHit.setFieldName(matchingField);
         suspectHit.setMatchComment(String.valueOf(totalScore).substring(0,totalScoreStringLength));
         suspectHitResults.add(suspectHit);
        }
        /*
        ** The fact that the match got as far as the cleared list is grounds for eliminating this name from future searches.
        ** This is useful because the name could come up again if another word matched.
        */
        matchingEntIds.add(entId);
       }
      }
     }
     else {
      //make note of the failure of these two words to match
      lastMatchingWord = matchingWord;
      //lastEntId = entId.intValue();
      lastWord = word;
     }
    }//End of for-loop
   }
   return suspectHitResults;
  }
  catch (Exception e) {
   e.printStackTrace();
   throw new PatriotSearchException();
   //return null;
  }
 }
 /*
 ** Set the hash table to null. This will force the search engine to reload the contents from database.
 ** This typically is called from the portlet after a suspect name has changed or been added.
 */
 public void invalidateQuickMatchHash() {
  _log.warn("force reload of suspects hashtable after data change");
  //LoaderImpl.quickMatches.clear();
  LoaderImpl.quickMatches = null;
 }
 /*
 * Check if the hit word was part of the surname. This is useful as it can be used to
 * weigh the score for the hit to a greater or lesser degree. In most cases a hit on a surname is
 * weighed more and a hit on a forename is appropriately weighed less. Currently this method only produces an
 * appropriate answer if the hit is to against an individual ("IND") type.
 * @param String unadulteratedMatchingName The complete name that was matched to
 * @param String hitWord The word that was hit. This method will determin if the word appears in the forename or in
the surname
 * @param String type Currently the method will only produce a positive or negative result if the type is 'IND' or
individual
 * @param String listType Not currently used.
 * @param intelligentSurnameMatching From filter.properties This must be the string "on" for the method to produce a
result
 * @return String  "undefined", "forenameHit" or "surnameHit".
 * @since 2.1
 */
 String hitOnSurname(String unadulteratedMatchingName, String hitWord, String type, String listType, String
intelligentSurnameMatching) {
  _log.info("hitOnSurname " + unadulteratedMatchingName + " hitWord " + hitWord + " type" + type + " listType " +
listType + " intelligentSurnameMatching " + intelligentSurnameMatching);
  String surnameHit = "undefined";
  /*
  ** intelligent surname matching is available only for certain types of match.
  ** Intelligent name matching can be turned on or off in the Filter.properties
  ** file.
  */
  //if (type.equals(Constants.INDIVIDUAL_TYPE_CODE) && listType.equals(Constants.SDN_LIST_TYPE_CODE) &&
intelligentSurnameMatching.equals("on")) {
  if (type.equals(Constants.INDIVIDUAL_TYPE_CODE) && intelligentSurnameMatching.equals("on")) {
   int surnameOffset = unadulteratedMatchingName.indexOf(",");
   if (surnameOffset > 0) {
    String surname =  unadulteratedMatchingName.substring(0, surnameOffset);
    if (surname != null && !surname.equals("")) {
     //Check if the surname contains the hitword
     if (surname.indexOf(hitWord) >= 0) {
      surnameHit = "surnameHit";
     _log.info("surname hit");
    }
    else {
     String forename = unadulteratedMatchingName.substring(surnameOffset + 1);
     //Check if the forename contains the hitword
     if (forename.indexOf(hitWord) >= 0) {
      surnameHit = "forenameHit";
      _log.info("forename hit");
     }
    }
   }
  }
 }
 _log.info("return from hitOnSurname " + surnameHit);
 return surnameHit;
}
/*
** This function is a duplicate of some code in the 'inClearedList' method. It will eventually replace that
** code. It's inclusion is for Cleared List searching which does not call 'inClearedList' but needs the details.
*/
private SuspectHitResult furnishHitDetails(String matchString, String suspectString, long entId, long altNum, long
originalWordInd, String listType, java.sql.Connection connX) {
 try {
  SuspectHitResult suspectHitResult = new SuspectHitResultImpl ();
  java.sql.Connection conn = DBConnection.getDBConnection();
  /*
  ** If the match was on an alias then return the name from the master list with the alias name concatenated.
  ** Slight differences in the two versions of matching name can occur because matchString has had all the
  ** irrelevent characters removed while the name has not. Remove the non-alphabetic characters.
  */
  String name = matchString;
  String keepName = name;
  name = removeNonAlphabeticCharacters(name.toUpperCase());
  if (!keepName.equalsIgnoreCase(matchString)) {
   if (originalWordInd != 1) {
    //True alias
    name = name + " (aka) " + matchString;
    suspectHitResult.setListFieldName("Alias");
   }
   else {
    //not a true alias, it's a hit on a translation of the original
    name = name + " (translation) " + matchString;
    suspectHitResult.setListFieldName("Name");
   }
  }
  else {
   suspectHitResult.setListFieldName("Name");
  }
  String aliasName = "";
  if (altNum > 0 && originalWordInd == Constants.TRANSLATION) {
   //Match was on a translation of an alias so get the original alias
   PreparedStatement pStatement = null;
   pStatement = conn.prepareStatement(_p.getProperty("sqlClearedListSqlName"));
   pStatement.setLong(1, entId);
   pStatement.setLong(2, altNum);
          pStatement.executeQuery();
          // Show result
          java.sql.ResultSet rsa =pStatement.getResultSet();
          while ( rsa.next() ) {//only one row expected
    aliasName = rsa.getString("alt_name").trim();
   }
  }
  suspectHitResult.setFieldName(keepName);
  //Tack on the original alias name if the hit was on a translation of the alias
  if (!aliasName.equals("")) {
   suspectHitResult.setFieldName(suspectHitResult.getFieldName() + " (translation of the alias) " + aliasName);
  }
  suspectHitResult.setFieldValue(suspectString);
  if (listType != null) {
   suspectHitResult.setListType(listType);
  }
  //RESOLVE - haven't got isSecure suspectHitResult.setListSecure(isSecureList);
  //If the hit was on a translation of an alias set the list field value to the original alias
  if (altNum > 0 && originalWordInd == Constants.TRANSLATION) {
   suspectHitResult.setListFieldValue(name + " of the alias " + aliasName);
  }
  else {
   //Set the name hit
   if (originalWordInd == Constants.TRANSLATION) {
    suspectHitResult.setListFieldValue(matchString + " (translation) ");
   }
   else {
    suspectHitResult.setListFieldValue(matchString);
   }
  }
  suspectHitResult.setEntId(entId);
  DBConnection.closeDBConnection(conn);
  return suspectHitResult;
 }
 catch ( Exception e ) {
        System.out.println( "Suspect Match Critical ErrorY: " + e);
  _log.error("Suspect Match Critical ErrorYY " + e);
       e.printStackTrace();
    }
 return null;
}
}
```